Leture notes for graduate course in Computer Science 50DT041 on

Computational Complexity Theory

Federico Pecora

Center for Applied Autonomous Sensor Systems
School of Science and Technology, Örebro University

federico.pecora@oru.se

Preface

This report summarizes the lectures given in the graduate course on Computational Complexity Theory (50DT041) given at Örebro University. This is an introductory graduate course aimed at PhD students whose background is not necessarily in Computer Science. The aim is to provide literacy in computational complexity and the ability to understand the complexity of new problems that may arise in one's own research. The course assumes as prerequisites topics typically covered in discrete math courses offered in Computer Science/Engineering departments. Also, it is assumed that students have taken the course "Topics in Contemporary Computer Science" (50DT056), which introduces models of computation, Turing machines, and the notion of decidability.

Chapter 2 of the book by Cormen et al. "Introduction to Algorithms" (MIT Press, 2014) is a good reference for Part I on Algorithm Complexity. The remainder of the course is based primarily on Chapters 17, 27, 28 and 29 of the book by Elaine Rich "Automata, Computability and Complexity" (Prentice Hall, 2008). Pointers to relevant sections in these books (referred to as CLRS and ER, respectively) are given in margin notes throughout this document. These lecture notes also include several additional topics which are not covered or treated only superficially in the two books, namely, the relation between decision and search problems, a more detailed discussion of the class coNP, and the complexity of problems with succinct representations.

These notes are intended to summarize and support the discussions carried out during class. They do not substitute reading the relevant sections of the books and attending classes. Note also that the material presented here is a constant work in progress.

For comments, errors, and omissions, please contact me at `federico.pecora@oru.se`. Special thanks go to the students of the fall 2018 edition of this course for their many comments and suggestions.

Federico Pecora
July 2019
Örebro, Sweden

Contents

Part I

Algorithm Complexity

1.1 An Initial Example

Let's consider the sorting problem:

Input: a sequence $A = \langle a_1, \ldots, a_n \rangle$ of numbers.

Output: a permulation $\pi(A) = \langle a'_1, \ldots, a'_n \rangle$ such that $a'_{i-1} \leq a'_i$ for all $i \in [2 \,..\, n]$.

Possible questions we may be interested in:

- Does there exist an algorithm to solve this problem?
- What computational resources do I need to run this algorithm?
- How long will it take to solve this problem?
- What features of the input determine how long it will take to solve this problem?
- Are there any "better" algorithms than the one I found? What is "better"?
- How hard is this problem in general?

This course

- provides the mathematical tools to answer these questions for any given problem;
- reveals how all these questions are related.

1.2 An Algorithm for Sorting

CLRS Chapter 2 Section 2.1

One of the most intuitive algorithms for sorting is Insertion Sort.

INSERTION-SORT(A)

```
1  for j = 2 to n
2      key = A[j]
3      i = j − 1
4      while i > 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i − 1
7      A[i + 1] = key
```

How efficient is this algorithm?

In order to answer this question, we first need to assume a model of computation (roughly speaking, the "computational framework" on which we can assume to "interpret" the pseudo-code above).

1.3 Random Access Machines

CLRS Chapter 2 Section 2.2

A Random Access Machine (RAM) is a model of computation with the following properties:

- Instructions are executed one after another, there is no concurrency
- Instructions are similar to those commonly found on real computers, that is
  - Arithemtic: add, subtract, multiply, divide, modulo, floor, ceiling, shift-left, shift-right, etc.
  - Data movement: load, store, copy
  - Control: conditional/unconditional branch, subroutine calls, return, for and while loops
- Any of the above instructions takes constant time
- We can represent words of at most $c \log n$ bits, where
  - n is the size of the input
  - $c \geq 1$ because we want to be able to hold the value of n and address its individual elements
  - c is constant because we cannot allow the word size to grow arbitrarily

1.4 Time Requirement of Insertion Sort

CLRS Chapter 2 Section 2.2

Analysis of INSERTION-SORT(A), where $A = \langle a_1, \ldots, a_n \rangle$:

- Line 1 is executed n times (the condition is true $n-1$ times, plus one time when it is false and the loop ends)
- Lines 2–3 happen $n-1$ times
- The number of times the condition in line 4 is true depends on the input A
  - Let the number of times line 4 is executed for a given j be t_j
  - So overall, line 4 is executed $\sum_{j=2}^{n} t_j$ times
  - Note this method of "hiding the hard part of the analysis under the carpet"
- Similarly, lines 5–6 are executed $\sum_{j=2}^{n}(t_j - 1)$ times
- Line 7 is executed $n-1$ times

So, in general, the time requirement for this procedure is

$$\begin{aligned}\text{timereq}(\text{INSERTION-SORT}(A)) = {} & c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j \\ & + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)\end{aligned}$$

Let's also assume that each operation takes not only constant time, but actually "one" time unit, that is:

$$\text{timereq}(\text{INSERTION-SORT}(A)) = 4n - 3 + \sum_{j=2}^{n} t_j + 2\sum_{j=2}^{n}(t_j - 1)$$

What is the overall temporal requirement of INSERTION-SORT(A) in the best case?

- In the best case, A is already sorted
- This is the best case for this algorithm (but not necessarily for others, see Quicksort for example) because
  - If A is sorted, then the number of times the condition in line 4 is true is minimized
  - That is, $t_j = 1$ for all $j \in [2\,..\,n]$
- Plugging in we get

$$\text{timereq}(\text{INSERTION-SORT}(A)) = 5n - 4$$

What is the overall temporal requirement of INSERTION-SORT(A) in the worst case?

- In the worst case, A is sorted in descending order
- This is the worst case for this algorithm (but not necessarily for others, see Quicksort for example) because
  - If A is in descending order, then the number of times line 4 is true is maximized
  - That is, $t_j = j$ for all $j \in [2\,..\,n]$
- Observing that

$$\sum_{j=1}^{n} j = [\text{arithmetic series}^1] = \frac{n(n+1)}{2}$$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n}(j-1) = [k = j-1] = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$$

[1] Recall Gauss' solution for computing the sum of the first N numbers (Hayes, 2006; Von Waltershausen, 1856).

- Plugging in we get

$$\text{timereq}(\textsc{Insertion-Sort}(A)) = \frac{3}{2}n^2 + 3n - 4$$

So, we get quadratic time requirement in the worst case, linear in the best case.

There are "better" algorithms, although the notion of "better" really depends on what you expect as input.

For example,

- Merge Sort runs in time proportional to $n \log n$ in all cases (best, worst, average)
- Quicksort runs in time proportional to n^2 in the worst case, but $n \log n$ in the average and best cases

1.5 Asymptotic Notation

CLRS Chapter 3 Section 3.1

But how do we represent a statement like "Insertion Sort runs in time proportional to n^2" mathematically?

To be precise, we use the following definitions.

1.6 Upper Bounds

CLRS Chapter 3 Section 3.1

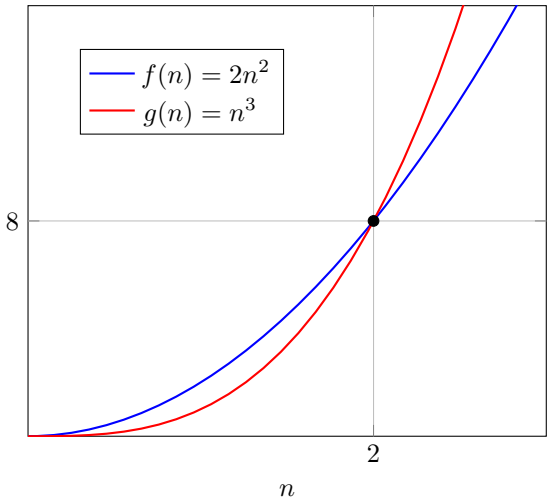
Aka "big-O" notation.

$$O(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$g(n)$ is an asymptotic upper bound for $f(n)$.

Note that $O(g(n))$ is the set of all functions that have this property.

Example: $2n^2 \in O(n^3)$, with $c = 1$ and $n_0 = 2$.



Other examples:

$$
\begin{aligned}
n^2 &\in O(n^2) \\
n^2 + n &\in O(n^2) \\
n^2 + 1000n &\in O(n^2) \\
1000n^2 + 1000n &\in O(n^2) \\
n &\in O(n^2) \\
n/1000 &\in O(n^2) \\
n^{1.9999} &\in O(n^2) \\
n^2/\log\log\log n &\in O(n^2)
\end{aligned}
$$

1.7 Lower Bounds

CLRS Chapter 3 Section 3.1

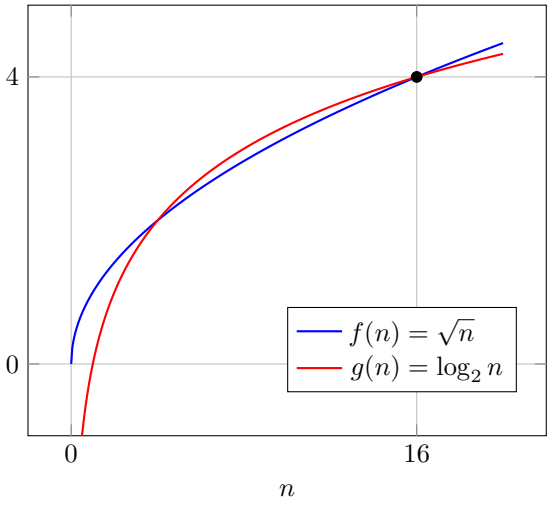
Aka "big-Omega" notation.

$$\Omega(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

$g(n)$ is an asymptotic lower bound for $f(n)$.

Note that $\Omega(g(n))$ is the set of all functions that have this property.

Example: $\sqrt{n} \in \Omega(\log n)$, with $c = 1$ and $n_0 = 16$.



Other examples:

$$
\begin{aligned}
n^2 &\in \Omega(n^2) \\
n^2 + n &\in \Omega(n^2) \\
n^2 - n &\in \Omega(n^2) \\
n^2 + 1000n &\in \Omega(n^2) \\
1000n^2 + 1000n &\in \Omega(n^2) \\
1000n^2 - 1000n &\in \Omega(n^2) \\
n^3 &\in \Omega(n^2) \\
n^{2.00001} &\in \Omega(n^2) \\
n^2 \log\log\log n &\in \Omega(n^2) \\
2^{2^n} &\in \Omega(n^2)
\end{aligned}
$$

1.8 Tight Bounds

CLRS Chapter 3 Section 3.1

Aka "big-Theta" notation.

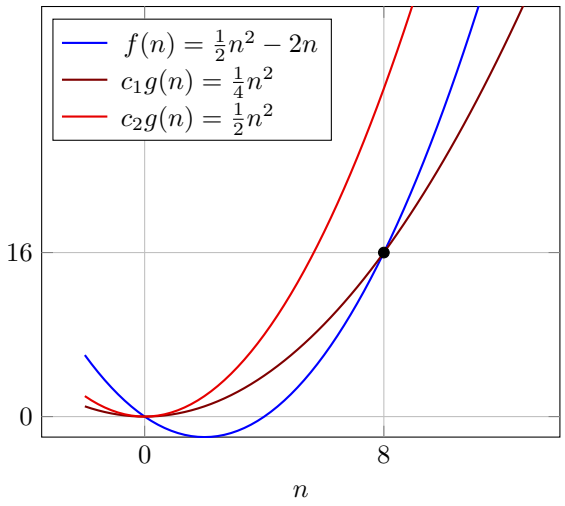
$$\Theta(g(n)) = \{f(n) : \exists c_1 > 0, c_2 > 0, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

$g(n)$ is an asymptotic tight bound for $f(n)$.

Note that $\Theta(g(n))$ is the set of all functions that have this property.

Theorem 1. *$f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.*

Example: $\frac{1}{2}n^2 - 2n \in \Theta(n^2)$, with $c_1 = \frac{1}{4}, c_2 = \frac{1}{2}$ and $n_0 = 8$.



1.9 Strict Upper and Lower Bounds

CLRS Chapter 3 Section 3.1

Similarly, we can define strict bounds (with alternative definitions in terms of limits):

$$o(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$
$$= \{f(n) : \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}$$

$$\omega(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$$
$$= \{f(n) : \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty\}$$

Note that $n^2 \notin o(n^2)$ and $n^2 \notin \omega(n^2)$.

1.10 Comparison of Functions

CLRS Chapter 3 Section 3.1

Relational properties:

- Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ implies $f(n) = \Theta(h(n))$
  - Same holds for O, Ω, o, ω
- Reflexivity: $f(n) = \Theta(f(n))$
  - Same holds for O, Ω
- Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
  - Same holds for o, ω

1.11 Comparing Growth Rates

Note a few important points about order of growth of common functions:

- $n^k \in o(n^m)$ for all $k < m$
- $\log_a n \in \Theta(\log_b n)$ for all $a > 1, b > 1$, because

$$\lim_{n\to\infty} \frac{\log_b n}{\log_a n} = \lim_{n\to\infty} \frac{\log_b a \log_a n}{\log_a n} = \log_b a$$

- $\log n \in o(n^k)$ for all $k > 0$, because

$$\lim_{n\to\infty} \frac{\log n}{n^k} = [\text{see above}] = \lim_{n\to\infty} \frac{\ln n}{n^k} = [\text{L'Hôpital's rule}] = \lim_{n\to\infty} \frac{n^{-1}}{kn^{k-1}} = \lim_{n\to\infty} \frac{1}{kn^k} = 0$$

- $n^k \in o(a^n)$ for all $a > 1, k > 0$, because

$$\lim_{n\to\infty} \frac{n^k}{a^n} = [\text{L'Hôpital's rule } k \text{ times}] = \lim_{n\to\infty} \frac{k!}{a^n (\ln a)^k} = 0$$

- $a^n \in o(b^n)$ for all $0 \leq a < b$, because

$$\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty} \left(\frac{a}{b}\right)^n = 0$$

- $a^n \in o(n!)$ for all $a > 0$, because

$$\lim_{n\to\infty} \frac{a^n}{n!} = \lim_{n\to\infty} \frac{\log(a^n)}{\log(n!)} = \lim_{n\to\infty} \frac{n \log a}{\sum_{i=1}^{n} \log i} = 0$$

1.12 Note on Exponential Functions

What is the difference between $O(2^n)$ and $2^{O(n)}$?

$$2^n \in O(2^n) \text{ and } 2^n \in 2^{O(n)}$$
$$2^{n^2} \notin O(2^n) \text{ and } 2^{n^2} \notin 2^{O(n)}$$
$$2^{\frac{n}{2}} \in O(2^n) \text{ and } 2^{\frac{n}{2}} \in 2^{O(n)}$$
$$3^n \notin O(2^n) \text{ but } 3^n \in 2^{O(n)}$$

In fact, $f(n) \in 2^{O(n)}$ iff

$$\log(f(n)) \in O(n),$$

which means that

$$\exists c > 0, n_0 > 0 \text{ such that } 0 \leq \log(f(n)) \leq cn \text{ for all } n \geq n_0,$$

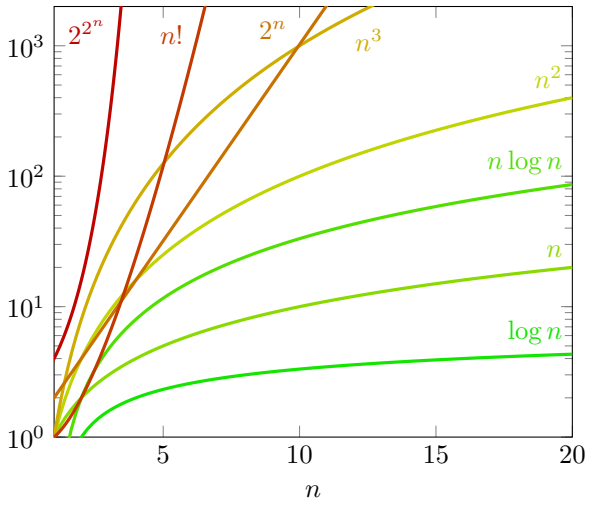
hence

$$\exists c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq 2^{cn} = (2^c)^n \text{ for all } n \geq n_0,$$

and hence that $f(n) \in O(b^n)$ for some $b > 0$ (similarly for Ω, ω, Θ, and o).

1.13 Common Functions in Complexity Analysis

Name	*Asymptotic notation*	*Examples*
logarithmic	$O(\log n)$	$\log n, \log(n^3)$
polylogarithmic	$\text{poly}(\log n)$	$\log n, (\log n)^2$
linear	$O(n)$	$n, 3n + 2$
quasi-linear	$n\,\text{poly}(\log n)$	$n \log n$ (linearithmic), $n(\log n)^3$
quadratic	$O(n^2)$	$n^2, 100n^2 + 2n + 1$
cubic	$O(n^3)$	$n^3, 0.3n^3$
polynomial	$2^{O(\log n)} = n^{O(1)} = \text{poly}(n)$	$n^3 + n, n^{1000}$
quasi-polynomial	$2^{\text{poly}(\log n)}$	$n^{(\log\log n)}, n^{(\log n)}$
sub-exponential	$2^{o(n)}$	$2^{n^{\frac{1}{5}}}, 2^{\sqrt{n}}$
exponential (linear exp.)	$2^{O(n)}$	$1.2^n, 42^n$
exponential	$2^{\text{poly}(n)}$	$2^n, 2^{n^3}$
factorial	$O(n!)$	$n!$
double exponential	$2^{2^{\text{poly}(n)}}$	2^{2^n}



1.14 Asymptotic Analysis of Recursive Functions

Let's implement the function $f(x, n) = x^n$. Formulated recursively,

$$f(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot f(x, n-1) & \text{otherwise} \end{cases}$$

EXP(x, n)
1 **if** $n = 0$
2 **return** 1
3 **return** $x \cdot$ EXP$(x, n - 1)$

Analysis of EXP(x, n):

- Line 1 is executed $n + 1$ times
- Line 2 is executed once
- Line 3 is executed n times

Hence, $\text{timereq}(\text{EXP}(x, n)) \in \Theta(n)$.

We can do better, thanks to a mathematician friend who tells us that:

$$f(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ (x^2)^{\frac{n}{2}} & \text{if } n \text{ is even} \\ x(x^2)^{\frac{(n-1)}{2}} & \text{if } n \text{ is odd} \end{cases}$$

Let's implement it:

```
ExpFast(x, n)
1  if n = 0
2      return 1
3  if n mod 2 = 0
4      return ExpFast(x · x, n/2)
5  return x · ExpFast(x · x, (n − 1)/2)
```

Analysis of ExpFast(x, n):

- Line 1 is executed as many times as there are recursive calls
- Line 2 is executed once
- One among line 3 or line 4 gets executed at each recursive call
- How many recursive calls are there goint to be?
- There will be k recursive calls, where
  - k is the number of times you can divide n by 2
  - That is, $2^k = n$, hence, $k = \log_2 n$

Hence, $\text{timereq}(\text{ExpFast}(x,n)) \in \Theta(\log n)$, which is much better than linear!

1.15 Best/Worst Case and Asymptotic Notation

So we have seen that $\text{timereq}(\text{ExpFast}(x,n)) \in \Theta(\log n)$ — it's Θ because we can provide a single function that serves as an upper and lower bound.

Recall that $\text{timereq}(\text{Insertion-Sort}(A)) \in \Theta(n^2)$ in the worst case, and $\text{timereq}(\text{Insertion-Sort}(A)) \in \Theta(n)$ in the best case — again, because we can provide a function that serves as both lower and upper bound, in both cases.

Note that $\text{timereq}(\text{Alg}) \in \Theta(f(n))$ does not mean that Alg has the same time requirement in all cases.

We are often interested in upper bounds, so we will most often use the "big-O" notation.

1.16 Time Requirement

ER Chapter 27 Section 27.4

The time requirement of a program "running" on a RAM is the total number of operators that are executed.

Henceforth,

- TM = deterministic Turing machine
- NDTM = non-deterministic Turing machine

How do we characterize the running time of a Turing Machine?

If M is a TM that halts on all inputs, then

$$\text{timereq}(M) = f(n) = \text{max. number of steps made on any input of length } n$$

If M is a NDTM all of whose computational paths halt on all inputs, then

$$\begin{aligned}\text{timereq}(M) = f(n) = &\\ &\text{max. number of steps made along}\\ &\text{any path executed on any input of length } n\end{aligned}$$

So, what's the relation between the Turing Machine model of computation and the RAM model of computation?

1.17 Equivalence Between Turing Machines and RAMs

ER Chapter 17 Section 17.4

We know that adding multiple tapes does not increase the power of TMs.

Neither does non-determinism.

What about adding features that would make a TM more like a real computer?

- Unbounded number of memory cells addressed by integers
- Instruction set of a RAM
- Program counter, address register, accumulator, special-purpose registers, input/output file

Can a TM simulate such a computer?

Theorem 2. *A RAM combined with a stored program can be simulated by a TM M. If the RAM program requires n steps to perform some operation, then* $\text{timereq}(M) \in O(n^6)$.

Proof. Constructs a 7-tape TM that simulates the computer, see (Rich, 2008) if interested. □

1.18 Seeing Problems as Languages

ER Chapter 17 Section 17.2

The SAT problem is the problem of verifying whether a formula in Boolean logic, e.g.,

$$w = (P \vee Q) \wedge (\neg P \vee Q),$$

has an assignment of Boolean values to variables that makes the formula true.

This can be seen as the problem of deciding whether a Boolean formula is a member of the language of all true Boolean formulae:

$$\text{SAT} = \{w : w \text{ is a true Boolean formula}\}$$

ER Chapter 27 Section 27.3.2

We will often "convert" optimization problems to decision problems as well.

For example, for the problem of

> Find the shortest path from vertex u to vertex v in a weighted undirected graph G

The corresponding decision problem could be defined as:

$$\begin{aligned}\text{SHORTEST-PATH} = \{(G, u, v, k) : &\ G \text{ is an undirected graph, } u \text{ and } v \text{ are vertices in } G,\\ &\ k \geq 0 \text{ and there exists a simple path from } u \text{ to } v\\ &\ \text{of length} \leq k\}\end{aligned}$$

Part II

The Language Class P

2.1 The Language Class P

ER Chapter 28 Section 28.1

$L \in$ P iff $\exists$ some deterministic Turing machine M that decides L, and $\text{timereq}(M) \in O(n^k)$ for some constant k.

We'll say that L is tractable iff it is in P.

2.2 Closure Under Complement

ER Chapter 28 Section 28.1.1

Theorem 3. *The class* P *is closed under complement.*

Proof. If M accepts L in polynomial time, swap accepting- and non-accepting states to accept $\neg L$ in polynomial time. □

But what is the complement exactly?

2.3 Defining Complement

ER Chapter 28 Section 28.1.1

$$\begin{aligned}\text{CONNECTED} &= \{G = (V, E) : G \text{ is an undirected graph and } G \text{ is connected}\}\\ \text{NOTCONNECTED} &= \{G = (V, E) : G \text{ is an undirected graph and } G \text{ is not connected}\}\\ \neg\text{CONNECTED} &= \text{NOTCONNECTED} \cup \{\text{strings that are not syntactically legal descriptions of undirected graphs}\}\end{aligned}$$

We know that CONNECTED $\in$ P (see later).

Hence, $\neg$CONNECTED $\in$ P by the closure theorem. What about NOTCONNECTED?

If we can check for legal syntax in polynomial time, then we can consider the universe of strings whose syntax is legal.

Then we can conclude that NOTCONNECTED belongs to P if $\neg$CONNECTED does.

2.4 Languages That Are in P

ER Chapter 28 Section 28.1.2

- Every regular language[2]
- Every context-free language[3] since there exist context-free parsing algorithms that run in $O(n^3)$ time
- Others, like $A^n B^n C^n$

2.5 Proving That a Language is in P

Since a RAM can be simulated by a TM in polynomial time, we can:

- Describe a TM that decides L in polynomial time, or
- State an algorithm for a conventional computer (hence, deterministic) that decides L in polynomial time

2.6 Example: Regular Languages

ER Chapter 28 Section 28.1.3

Theorem 4. *Every regular language is in* P.

[2] Recall: Regular language = language recognized by Deterministic Finite State Machine (DFSM) / regular expressions.

[3] Recall: CF language = production rules are $1:1$, $1:n$, or $1:0$. Compare with context-sensitive grammars (not in P), where left-hand side can be surrounded by context of terminal and non-terminal symbols.

Proof. Every regular language can be decided in linear time, as, if L is regular, there exists some Deterministic Finite State Machine (DFSM) M that decides it. Construct a deterministic TM M' that simulates M, moving its read/write head one square to the right at each step. When M' reads a terminal character, it halts. If it is in an accepting state, it accepts; otherwise it rejects. On any input of length n, M' will execute $n + 2$ steps. Hence, $\text{timereq}(M') \in O(n)$. □

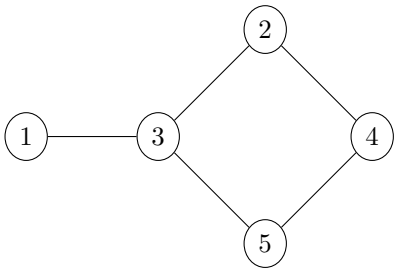
2.7 Example: Context-Free Languages

ER Chapter 28 Section 28.1.3

Theorem 5. *Every context-free language is in* P.

Proof. The Cocke-Kasami-Younger (CKY) algorithm can parse any context-free language in time that is $O(n^3)$ if we count operations on a conventional computer. That algorithm can be simulated on a standard, one-tape Turing machine in $O(n^{18})$ steps. Hence, every context-free language can be decided in $O(n^{18})$ time. □

2.8 Graph Languages



We can represent a graph $G = (V, E)$ as:

- Number of vertices followed by list of vertex-pairs (edges)
  - in the example: 101/1/11/11/10/10/100/100/101/11/101
  - requires string of length $O(|V|^2 \log_2 |V|)$, since there are at most $|V|^2$ edges
- Or as an adjacency matrix
  - in the example:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

  - requires string of length $\Theta(|V|^2)$

2.9 Connected Graphs

ER Chapter 28 Section 28.1.4

$$\text{CONNECTED} = \{G = (V, E) : G \text{ is an undirected graph and } G \text{ is connected}\}$$

Is CONNECTED in P?

CONNECTED$(G = (V, E))$

```
1  Set all vertices to be unmarked
2  Select a vertex v
3  L = {v}
4  n_marked = 1
5  while L ≠ ∅
6        v = POP(L)
7        for (v, u) ∈ E
8              if u not marked
9                    Mark u
10                   L = L ∪ {u}
11                   n_marked = n_marked + 1
12 if n_marked = |V|
13       return TRUE
14 return FALSE
```

Analysis of CONNECTED(G)

- Line 1 takes $O(|V|)$
- Lines 2–4 each take constant time
- The condition in line 5 is checked at most $|V|$ times
  - Line 6 takes constant time
  - Condition in line 7 is executed at most $|E|$ times
  - Lines 8–11 each take constant time
- Lines 12 and 13 takes constant time

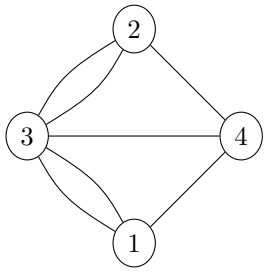
So, $\text{timereq}(\text{CONNECTED}(G)) = |V| \cdot O(|E|) = O(|V| \cdot |E|)$.

Note that $|E| \leq |V|^2$, so $\text{timereq}(\text{CONNECTED}(G)) \in O(|V|^3)$.

2.10 Eulerian Paths and Circuits

ER Chapter 28 Section 28.1.5

Seven Bridges of Königsberg (modern-day Kaliningrad) — represented as a graph:



Eulerian path through a graph $G = (V, E)$: a path that traverses each edge in E exactly once.

Eulerian circuit through a graph $G = (V, E)$: a path that starts at some vertex $s \in V$, ends back in s, and traverses each edge in E exactly once.

$$\text{EULERIAN-CIRCUIT} = \{G : G \text{ is an undirected graph and } G \text{ contains a Eulerian circuit}\}$$

Why is this useful? Bridge inspectors, road cleaners, and network analysts can minimize their effort if they traverse their systems by following a Eulerian path.

Is EULERIAN-CIRCUIT in P?

2.11 Euler Observes...

ER Chapter 28 Section 28.1.5

Degree of a vertex: number of edges with it as an endpoint.

A connected graph possesses a Eulerian path that is not a circuit iff it contains exactly two vertices of odd degree. Those two vertices will serve as the first and last vertices of the path.

A connected graph possesses a Eulerian circuit iff all its vertices have even degree. Because each vertex has even degree, any path that enters it can also leave it without reusing an edge.

So now we can state an algorithm for deciding EULERIAN-CIRCUIT:

```
EULERIAN(G = (V, E))
1  if ¬CONNECTED(G)
2        return FALSE
3  for v ∈ V
4        n_v = |{(u, v) ∈ E : u ≠ v}|
5        if n_v is odd
6              return FALSE
7  return TRUE
```

Analysis of EULERIAN(G):

- We have shown that connected runs in $O(|V|^3)$ time.
- The condition in line 3 is evaluated at most $|V|$ times.
  - Line 4 requires time that is $O(|E|)$.
  - Lines 5–6 require constant time.
- Line 7 takes constant time.

Hence, $\text{timereq}(\text{EULERIAN}(G)) \in O(|V|^3)$.

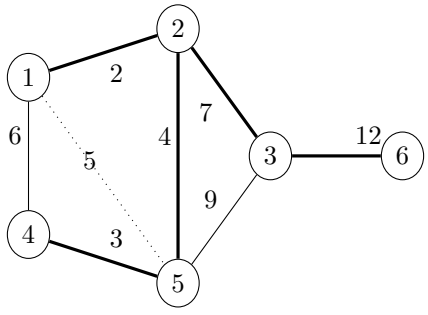
2.12 Spanning Trees

ER Chapter 28 Section 28.1.6

A spanning tree T of a graph $G = (V, E)$ is a graph whose edges are a subset of E, such that:

- T contains no cycles, and
- Every vertex in G is connected to every other vertex using just the edges in T.

Bold edges below constitute a spanning tree, dotted edge is alternative to $(2, 5)$.



A connected graph G will have at least one spanning tree; it may have many.

2.13 Minimum Spanning Trees

ER Chapter 28 Section 28.1.6

A weighted graph is a graph that has a weight associated with each edge.

An unweighted graph is a graph that does not associate weights with its edges.

If G is a weighted graph, the cost of a spanning tree is the sum of the costs (weights) of its edges.

A tree T is a minimum spanning tree of G iff

- it is a spanning tree, and
- there is no other spanning tree whose cost is lower than that of T.

$$\text{MST} = \{(G, c) : G \text{ is an undirected graph with positive cost on each edge and} \\ \exists \text{ minimum spanning tree of } G \text{ with total cost} \leq c\}$$

Relevance:

- Cheapest way to lay cables between points
- Bridge inspection

Is MST in P?

2.14 Kruskal's Algorithm

ER Chapter 28 Section 28.1.6

KRUSKAL$(G = (V, E))$

```
1  Sort the edges in E in ascending order by their cost (break ties arbitrarily)
2  Initialize T to a forest with an empty set of edges
3  while not all edges in E have been considered
4        (u, v) = the next edge in E
5        if u and v are not connected in T
6              T = T ∪ {(u, v)}
7  return T
```

If G is not connected, then KRUSKAL(G) finds a minimum spanning forest[4] (a minimum spanning tree for each connected component).

If G is connected, then KRUSKAL(G) finds a minimum spanning tree.

2.15 MST is in P

ER Chapter 28 Section 28.1.6

We can use KRUSKAL(G) to solve MST(G, c) as follows:

- Run KRUSKAL(G) to obtain minimum spanning tree T
- $c(T) =$ sum of weights in T
- If $c(T) \leq c$ accept, else reject

Analysis of KRUSKAL(G):

- Line 1 (sorting) takes $O(|E| \log |E|)$ with Merge Sort
- Line 2 takes constant time
- The condition in line 3 is checked $O(|E|)$ times
  - Line 4 takes constant time
  - Checking the condition in line 5 takes $O(|V|)$ time (may go through all other vertices)
    Line 6 takes constant time
- Line 7 takes constant time

So, $\text{timereq}(\text{KRUSKAL}(G)) \in O(|E| \log |E| + |E| \cdot |V|) = O(|E| \cdot |V|)$.

With a more efficient implementation of line 3, it is possible to show that it is also $O(|E| \log |V|)$.

[4] Forest = an undirected graph, all of whose connected components are trees = a disjoint union of trees = an undirected acyclic graph.

Part III

The Language Class NP

3.1 The Traveling Salesperson Problem (TSP)

ER Chapter 28 Section 28.2.3

Hamiltonian path through a graph G: a path that traverses each vertex in G exactly once.

Hamiltonian circuit through a graph G: a path that starts at some vertex s, ends back in s, and traverses each other vertex in G exactly once.

Traveling Salesperson Problem (TSP): given a weighted graph $G = (V, E)$, find a Hamiltonian circuit (starting and ending in some vertex s) with lowest cost.

ER Chapter 27 Section 27.1

The corresponding decision problem can be stated as follows:

$$\text{TSP-DECIDE} = \{(G, c) : G \text{ is an undirected graph with positive edge weights and } G \text{ contains a Hamiltonian circuit whose cost} \leq c\}$$

We can easily write a NDTM that decides TSP-DECIDE:

TSP-DECIDE$(G = (V, E), c)$

1 Create an empty sequence of vertices S
2 $W = V$
3 **while** $W \neq \emptyset$
4 $\quad v =$ CHOOSE(W)
5 $\quad W = W \setminus \{v\}$
6 $\quad$ Add v to sequence S
7 **if** S is a Hamiltonian circuit and sum of costs along $S \leq c$
8 $\quad$ **return** TRUE
9 **return** FALSE

The choice in line 4 is an example of non-determininstically deciding.

NDTMs branch into many copies, each following one possible transition.

TMs have a single computation path.

NDTMs have multiple computation paths.

Recall: Can NDTMs solve problems that TMs cannot? No, NDTMs are only more efficient!

3.2 The Language Class NP

ER Chapter 28 Section 28.2

$L \in$ NP iff

- there is some NDTM M that decides L, and
- $\text{timereq}(M) \in O(n^k)$ for some constant k.

3.3 Certificates and Verifying

A TM V is a verifier for a language L iff

$$w \in L \text{ iff } \exists p : (\langle w, p \rangle \in L(V))$$

We call p a certificate: this contains all the necessary information to verify that a given w decides the decision problem L.

As an example, a certificate for TSP would contain the graph G and cost c as well as evidence of a Hamiltonian Circuit in the form of a sequence of veritces:

$$p = \langle G = (V, E), c, v_1, v_2, \ldots, v_n \rangle$$

3.4 The Relation Between Verifying and Deciding

ER
Chapter 28
Section 28.2.1

An alternative definition for the class NP is the following.

$L \in$ NP iff there exists a deterministic TM V such that:

- V is a verifier for L, and
- $\text{timereq}(V) \in O(n^k)$ for some constant k

Theorem 6. *These two definitions are equivalent:*

Def. 1: $L \in$ NP *iff there exists a polynomial-time NDTM that decides it.*
Def. 2: $L \in$ NP *iff there exists a polynomial-time TM that verifies it.*

Proof. Let $L \in$ NP by Def. 1.

- There exists NDTM M that decides L in polynomial time
- We construct TM V that can verify L in polynomial time:
  - On input $\langle w, p \rangle$, V simulates M running on w, except that
  - Every time M would make a choice, V follows the "path" given by the next symbol in p
- V accepts iff M would have accepted on path p
- Thus, V accepts iff p is a certificate for w
- V runs in polynomial time because the length of the longest path M can follow is bounded by some polynomial function over the length of w (that is, M takes polynomial time, as per Def. 1)

Let $L \in$ NP by Def. 2.

- There exists a TM V such that V is a verifier for L and $\text{timereq}(V) \in O(n^k)$ for some k
- We construct NDTM M that decides L in polynomial time:
  - On input w, M non-deterministically selects a certificate p
  - Any certificate p need not be longer than the max steps it would take V to verify it, which is polynomial by Def. 2
  - M then runs V on $\langle w, p \rangle$
- M will follow a finite number of paths, each halting in $O(n^k)$, hence it is a polynomial decider for L

□

Summarizing, we can see the class NP as follows:

- NP is the class of problems that have succinct qualifying certificates
  - This certificate is an accepting path of a NDTM, which can only be polynomial in size because it took one computation path polynomial time to write it
- NP is the class of problems for which a qualifying certificate can be checked efficiently
  - Because there is a verifying TM that runs in polynomial time

3.5 Proving That a Language is in NP

Now we know that there are two ways to do this:

- Exhibit an NDTM to decide it, or
- Exhibit a TM to verify it

3.6 TSP-DECIDE is in NP (via decision procedure)

If G has n nodes, $\text{timereq}(\text{TSP-DECIDE}(G, c)) \in O(n)$.

Hence, we can exhibit a NDTM that decides the problem in polynomial time.

Hence, the decision problem TSP-DECIDE belongs to the class NP.

3.7 TSP-DECIDE is in NP (via verification procedure)

Suppose an oracle provides a certificate p for a given (G, c)

$$p = \langle G = (V, E), c, v_1, v_2, \ldots, v_n \rangle$$

How long would it take to verify whether p proves that $(G, c) \in$ TSP-DECIDE? Obviously, polynomial time via the following deterministic algorithm.

TSP-VERIFY(p)
1 **if** $n \neq |V|$
2 **return** FALSE
3 **if** $v_1 \neq v_n$
4 **return** FALSE
5 **if** $(v_i, v_{i+1}) \notin E$ for some $i \in [1, n-1]$
6 **return** FALSE
7 **if** sum of weights along path is $> c$
8 **return** FALSE
9 **return** TRUE

3.8 Boolean Satsifiability (SAT)

ER Chapter 22 Section 22.4.1

A wff w is a well-formed-formula in Boolean logic.

$$\text{SAT} = \{w : w \text{ is a Boolean wff and } w \text{ is satisfiable}\}$$

Examples:

$$w_1 = P \wedge Q \wedge \neg R$$
$$w_2 = P \wedge Q \wedge R$$
$$w_3 = P \wedge \neg P$$
$$w_4 = P \vee \neg P$$
$$w_5 = P \wedge (Q \wedge \neg R) \wedge \neg Q$$
$$w_6 = (P \vee Q) \wedge (Q \vee \neg R) \wedge \neg R$$

A literal is either a variable or a variable preceded by a single negation symbol.

3.9 SAT is in NP (via decision procedure)

ER Chapter 28 Section 28.2.5

Can we write a non-deterministic procedure for deciding SAT in polynomial time?

SAT-DECIDE(w)
1 **for** each variable v in w
2 CHOOSE$(\{\top, \bot\})$ and assign it to v
3 **if** EVAL(w)
4 **return** TRUE
5 **return** FALSE

Analysis of SAT-DECIDE(w):

- Lines 1 and 2 happen #variables times and take constant time
- Line 3 happens once and takes $O(\#\text{operators})$

Hence, SAT $\in$ NP.

3.10 SAT is in NP (via verification procedure)

ER Chapter 28 Section 28.2.5

Can we write a deterministic procedure for verifying a complete assignment a in polynomial time?

SAT-VERIFY($\langle w, a \rangle$)
1 **for** each variable v in w
2 Assign value prescribed in a to v
3 Evaluate the formula

Analysis of SAT-VERIFY($\langle w, a \rangle$):

- Lines 1 and 2 happen #variables times and take constant time
- Line 3 happens once and takes O(#operators)

Hence, SAT $\in$ NP.

3.11 3-SAT

ER Chapter 28 Section 28.2.5

A clause is either a single literal or the disjunction of two or more literals.

A wff is in conjunctive normal form (or CNF) iff it is either a single clause or the conjunction of two or more clauses.

A wff is in 3-conjunctive normal form (or 3-CNF) iff it is in conjunctive normal form and each clause contains exactly three literals.

Well-formed-formula (wff)	*3-CNF*	*CNF*
$(P \vee \neg Q \vee R)$	✓	✓
$(P \vee \neg Q \vee R) \wedge (\neg P \vee Q \vee \neg R)$	✓	✓
P		✓
$(P \vee \neg Q \vee R \vee S) \wedge (\neg P \vee \neg R)$		✓
$P \Rightarrow Q$		
$(P \wedge \neg Q \wedge R \wedge S) \vee (\neg P \wedge \neg R)$		
$\neg(P \vee Q \vee R)$		

Every wff can be converted to an equivalent wff in CNF in polynomial time.

$$\text{3-SAT} = \{w : w \text{ is a Boolean wff, } w \text{ is in 3-CNF, and } w \text{ is satisfiable}\}$$

Is 3-SAT in NP? Yes (same argument as for SAT).

(Is 2-SAT in NP? Yes, but we will see that it is not "as hard" as other problems in NP.)

3.12 Other Languages in NP

ER Chapter 28 Section 28.2.2

$$\text{HAMILTONIAN-PATH} = \{G : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian path}\}$$

$$\text{HAMILTONIAN-CIRCUIT} = \{G : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian circuit}\}$$

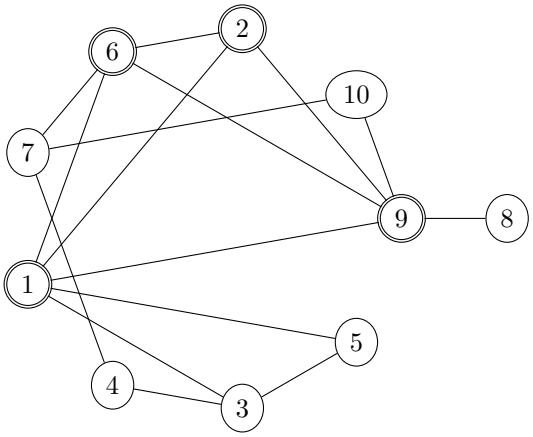
$$\text{TSP-DECIDE} = \{(G, c) : G \text{ is an undirected graph with positive edge weights and } G \text{ contains a Hamiltonian circuit with cost} \leq c\}$$

3.13 Cliques

ER Chapter 28 Section 28.2.4

A clique in G is a subset of V where every pair of vertices in the clique is connected by some edge in E.

A k-clique is a clique that contains exactly k vertices.



$$\text{CLIQUE} = \{(G, k) : G \text{ is an undirected graph with vertices } V \text{ and edges } E,\\ k \text{ is an integer, } 1 \leq k \leq |V|, \text{ and}\\ G \text{ contains a } k\text{-clique}\}$$

Relevance:

- Social sciences (sets of people who know each other)
- Biology (ecological niches)
- Test pattern generation (a large clique in an incompatibility graph of possible faults provides a lower bound on the size of a test set)

3.14 Graph Isomorphism

Two graphs G_1 and G_2 are isomorphic to each other iff there exists a way to rename the vertices of G_1 so that the result is equal to G_2 (i.e., iff their drawings are identical except for the labels on the vertices).

$$\text{SUBGRAPH-ISOMORPHISM} = \{(G_1, G_2) : G_1 \text{ is isomorphic to some subgraph of } G_2\}$$

Relevance:

- Chemistry, isomorphism among compounds.
- Scene understanding, isomorphism among scene descriptors.

3.15 Shortest Substrings

$$\text{SHORTEST-SUPERSTRING} = \{(S, k) : S \text{ is a set of strings and } \exists \text{ some superstring } T\\ \text{s.t. every } s \in S \text{ is a substring of } T \text{ and } |T| \leq k\}$$

Relevant in DNA seqencing.

3.16 Subset Sums

ER Chapter 28 Section 28.2.2

A multiset is a sets in which duplicates are allowed.

$$\text{SUBSET-SUM} = \{(S, k) : S \text{ is a multiset of integers and}\\ \exists \text{ some subset of } S \text{ whose elements sum to } k\}$$

Examples:

- $(\{1256, \underline{45}, \underline{1256}, 59, \underline{34687}, 8946, 17664\}, 35988) \in$ SUBSET-SUM
- $(\{101, 789, 5783, 6666, 45789, 996\}, 29876) \notin$ SUBSET-SUM

Relevant in cryptography:

- Given $f : S \mapsto 2^{\mathbb{N}}$ (strings to sets of integers)
- Storage of password p: store $\sum_{i \in f(p)} i$ instead of password
- Verification of user-given password p': check that $\sum_{i \in f(p')} i = \sum_{i \in f(p)} i$
- If hackers steal $\sum_{i \in f(p)} i$ they need to solve SUBSET-SUM to get p

3.17 Set Partitioning

ER Chapter 28 Section 28.2.2

$$\text{SET-PARTITION} = \{S : S \text{ is a multiset of objects, each with an associated cost, and } S \text{ can be divided into } (A, S \setminus A) \text{ s.t. } \sum_{i \in A} i = \sum_{i \in S \setminus A} i\}$$

Relevance:

- Like KNAPSACK without values
- Load balancing (cost = time to produce something)

3.18 Knapsack

ER Chapter 28 Section 28.2.2

$$\text{KNAPSACK} = \{(S, v, c) : S \text{ is a set of objects, each with an associated cost and value, and there is some way of choosing elements of } S \text{ (duplicates allowed) s.t. the total cost of the chosen objects} \leq c \text{ and their total value} \geq v\}$$

Relevance: thieves, backpackers, choosing anything that has cost and adds value, …

3.19 Bin Packing

ER Chapter 28 Section 28.2.2

$$\text{BIN-PACKING} = \{(S, c, k) : S \text{ is a set of objects each with associated size and } S \text{ can be divided so that objects fit into } k \text{ bins, each of which has size } c\}$$

Relevance:

- Packing boxes into containers (3D)
- Packing tiles/windows on a screen (2D)

3.20 Relation Between P and NP

ER Chapter 28 Section 28.3

Wait a moment: can't we make a

- Deterministic polynomial-time verifier, or
- Non-deterministic polynomial-time decider

for the sorting problem?

Of course! The decision problem associated to sorting does belong in NP. In fact,

Theorem 7. P $\subseteq$ NP.

Proof. Let $L \in$ P. Then there exists TM M that decides L in polynomial time. But M is also a nondeterministic decider for L (it just doesn't have to guess), hence $L \in$ NP as well. □

Is P $\subsetneq$ NP? Stay tuned…

3.21 Polynomial-Time Reductions

ER Chapter 28 Section 28.4

A mapping reduction R from L_1 to L_2 is a TM that implements some computable function f with the property that:

$$\forall x(x \in L_1 \Leftrightarrow f(x) \in L_2)$$

Suppose there exists a TM M that decides L_2.

Then, to decide whether $x \in L_1$ we can apply R to x and then invoke M to decide membership in L_2.

So, $C(x) = M(R(x))$ will decide L_1.

If R is a deterministic, polynomial-time procedure, then we say that L_1 is deterministic, polynomial-time reducible to L_2, that is:

$$L_1 \leq_P L_2$$

3.22 Using Reduction in Complexity Proofs

ER Chapter 28 Section 28.4

If $L_1 \leq_P L_2$ then:

- L_1 must be in P if L_2 is
  - If $L_2 \in$ P then $\exists$ polynomial-time TM M that decides it.
  - So, $M(R(x))$ is also a polynomial-time TM and it decides L_1.
- L_1 must be in NP if L_2 is
  - If $L_2 \in$ NP then $\exists$ polynomial-time NDTM M that decides it.
  - So, $M(R(x))$ is also a polynomial-time NDTM and it decides L_1.

3.23 Why Use Reduction?

ER Chapter 28 Section 28.4

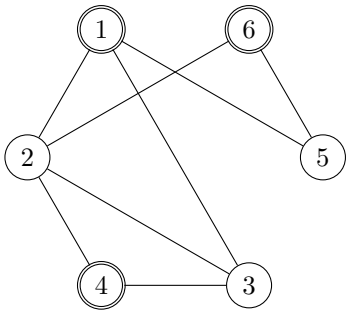
Given $L_1 \leq_P L_2$, we can use reduction to:

- Prove that $L_1 \in$ P or $L_1 \in$ NP because we already know that L_2 is.
- Prove that L_1 would be in P or in NP if we could somehow show that L_2 is
  - Allows to cluster languages of similar complexity (even if we're not yet sure what that complexity is).
  - In other words, L_1 is no harder than L_2 is.

3.24 The INDEPENDENT-SET Problem

ER Chapter 28 Section 28.4

Given $G = (V, E)$ an independent set of vertices is such that no two vertices are adjacent (i.e., connected by a single edge).



$$\text{INDEPENDENT-SET} = \{(G, k) : G \text{ is an undirected graph and } G \text{ contains an independent set of at least } k \text{ vertices}\}$$

Relevance in scheduling:

- Vertices are tasks
- Edges are task conflicts
- Largest number of tasks that can be scheduled at the same time = largest independent set

3.25 3-SAT and INDEPENDENT-SET

ER Chapter 28 Section 28.4

Strings in 3-SAT describe formulas that contain literals and clauses:

$$s_{\text{3-SAT}} = (P \vee Q \vee \neg R) \wedge (R \vee \neg S \vee Q)$$

Strings in INDEPENDENT-SET describe graphs that contain vertices and edges:

$$s_{\text{INDEPENDENT-SET}} = 101/1/11/11/10/10/100/100/101/11/101$$

We will explore the reduction 3-SAT $\leq_P$ INDEPENDENT-SET.

3.26 Gadgets

A gadget is a structure in L_2 (the target language, INDEPENDENT-SET) that mimics the role of a corresponding structure in L_1 (the source language, 3-SAT):

$$s_{\text{3-SAT}} \xrightarrow{\text{gadget}} s_{\text{INDEPENDENT-SET}}$$

So we need two gadgets:

- a gadget that looks like a graph but that mimics a literal, and
- a gadget that looks like a graph but that mimics a clause

3.27 3-SAT $\leq_P$ INDEPENDENT-SET

ER Chapter 28 Section 28.4

Let w be a CNF wff with k clauses.

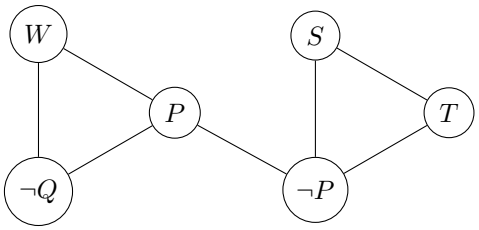
$R(w)$ is defined as follows:

1. Build a graph G
   (a) Create one vertex for each instance of each literal in w
   (b) Create an edge between each pair of vertices representing literals in the same clause

(c) Create an edge between each pair of vertices for complementary literals

2. Return (G, k)

For example, if $w = (P \vee \neg Q \vee W) \wedge (\neg P \vee S \vee T)$, then $R(w)$ would be the following graph:



3.28 R is Correct

ER Chapter 28 Section 28.4

We need to show that

1. $w \in$ 3-SAT $\Rightarrow R(w) \in$ INDEPENDENT-SET, and
2. $R(w) \in$ INDEPENDENT-SET $\Rightarrow w \in$ 3-SAT

Proving 1.

- There is a satisfying assignment A to the symbols in w
  - Hence, each clause has at least one literal made positive by A
- Is $R(w) \in$ INDEPENDENT-SET?
- That is, is there a subset S of k vertices of G that is an independent set?
- We can build S as follows
  (a) From each clause gadget choose one literal that is made positive by A
  (b) Add the vertex corresponding to that literal to S
- S contains exactly k vertices
  - We chose one vertex for each of the k clauses
- S is an independent set
  - No two vertices come from the same clause, so there cannot be an edge between them
  - No two vertices correspond to complimentary literals, so there cannot be an edge between them

Proving 2.

- $R(w) = G$ contains an independent set S of size k
- Is there some satisfying assignment A for w?
- No two vertices in S come from the same clause gadget (as otherwise they would be connected with an edge)
- Since S contains at least k vertices and w contains k clauses, then S must contain one vertex from each clause
- Build A as follows
  (a) Assign $\top$ to each literal that corresponds to a vertex in S
  (b) Assign arbitrary values to all other literals
- Since each clause will contain at least one literal whose value is $\top$, the value of w will be $\top$

3.29 Why Do Reductions?

ER Chapter 28 Section 28.5.1

Would we ever choose to solve 3-SAT by reducing it to INDEPENDENT-SET? Perhaps, if we had an efficient solver for INDEPENDENT-SET.

But that's not why we have introduced reductions.

A language L might have these properties:

- Property 1. $L \in \mathsf{NP}$
- Property 2. $L' \leq_P L$ for all $L' \in \mathsf{NP}$

L is NP-hard iff it possesses Property 2.

L is NP-complete iff it possesses both Property 1 and Property 2.

An NP-hard language is at least as hard as any other language in NP.

All NP-complete languages can be viewed as being equivalently hard.

3.30 NP-Completeness and P

If any NP-complete language is also in P, then all of them are and $\mathsf{P} = \mathsf{NP}$.

3.31 Relation Between P and NP (again)

In practice, temporal complexity is strongly curtailed by non determinism.

We strongly believe that $\mathsf{P} \neq \mathsf{NP}$.

The consequences of proving $\mathsf{P} = \mathsf{NP}$ would be huge:

- Efficient algorithms would exist for all problems in NP
- Most cryptography systems would break
- We could automatically prove any theorem which has a proof of reasonable length
- Many of the complexity classes would collapse into one

3.32 Recall a Problem in Class P

$$\text{EULERIAN-CIRCUIT} = \{G : G \text{ is an undirected graph and } G \text{ contains a Eulerian circuit}\}$$

We know that EULERIAN-CIRCUIT $\in$ P and that EULERIAN-CIRCUIT $\in$ NP.

But we cannot prove that EULERIAN-CIRCUIT is NP-hard, as there are plenty of problems $L' \in \mathsf{NP}$ for which we don't have a reduction $L' \leq_P$ EULERIAN-CIRCUIT.

That's what makes this problem "less hard" than, say, INDEPENDENT-SET (as we will see, we can show that the latter is NP-hard, and hence, NP-complete).

3.33 Example: Sudoku

ER
Chapter 28
Section 28.5.1

Rules of Sudoku: every line, column and square should contain all the digits 1–9

$$\text{SUDOKU} = \{b : b \text{ is a configuration of an } n \times n \text{ grid and } b \text{ has a solution under the rules of Sudoku}\}$$

	5			9	4	2	1	
4							8	
	3		7					
				2				4
2			4		6			8
6				3				
					8		6	
	7							3
	6	8	9	5			4	

A deterministic, polynomial-time verifier for SUDOKU, given the certificate:

$$\langle b, (\text{string representation of full assignment of } b)\rangle$$

- For each row, check that all numbers 1–9 appear exactly once
- For each column, check that all numbers 1–9 appear exactly once
- For each square, check that all numbers 1–9 appear exactly once

Clearly, requires $O(n^2)$ time.

So, SUDOKU $\in$ NP.

3.34 Example: Chess

ER Chapter 28 Section 28.5.1

$$\text{CHESS} = \{b : b \text{ is a configuration of an } n \times n \text{ chess board and there is a guaranteed win for the current player}\}$$

A deterministic, polynomial-time verifier for CHESS?

Any certificate would have to be a policy prescribing how the current player should move given the possible moves of the other player.

We could think of verifying this with a non-deterministic procedure in polynomial time… but it's hard to imagine a polynomial-time, deterministic procedure!

CHESS is therefore not known to be in NP.

3.35 Showing that L is NP-Complete

ER Chapter 28 Section 28.6.2

We would need to show that all languages in NP can be reduced in polynomial time to L — clearly infeasible.

But suppose we had one language L' that we know is NP-complete.

Then, we could show that any language L is NP-complete by finding a polynomial-time mapping reduction R from L' to L.

In other words, L is NP-complete iff

- Property 1. $L \in$ NP, and
- Property 2. $\exists L'$ such that $L' \leq_P L$ and L' is NP-complete

3.36 Finding an L' That is NP-Complete

ER Chapter 28 Section 28.5.2

The key property that every NP language has is that it can be decided by a polynomial-time NDTM.

So we need a language in which we can describe computations of NDTMs.

This language is

$$\text{SAT} = \{w : w \text{ is a Boolean wff and } w \text{ is satisfiable}\}$$

Theorem 8 (Cook-Levin Theorem)**.** *SAT is* NP*-complete.*

Proof. We've done half of it already (albeit the easy half):

- SAT $\in$ NP because we have shown a non-deterministic, polynomial-time procedure to decide it (as well as a deterministic, polynomial-time procedure to verify certificates)
- Prove that SAT is NP-hard (actual construction of a NDTM that decides SAT)

□

3.37 NP-Complete Languages

ER Chapter 28 Section 28.6

$$\text{SUBSET-SUM} = \{(S, k) : S \text{ is a multiset of integers and } \exists \text{ some subset of } S \text{ whose elements sum to } k\} \text{ is NP-complete}$$

$$\text{SET-PARTITION} = \{S : S \text{ is a multiset of objects, each with an associated cost, and } S \text{ can be divided into } (A, S \setminus A) \text{ s.t. } \sum_{i \in A} i = \sum_{i \in S \setminus A} i\} \text{ is NP-complete}$$

$$\text{KNAPSACK} = \{(S, v, c) : S \text{ is a set of objects, each with an associated cost and value, and there is some way of choosing elements of } S \text{ (duplicates allowed) s.t. the total cost of the chosen objects} \leq c \text{ and their total value} \geq v\} \text{ is NP-complete}$$

$$\text{HAMILTONIAN-PATH} = \{G : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian path}\} \text{ is NP-complete}$$

$$\text{HAMILTONIAN-CIRCUIT} = \{G : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian circuit}\} \text{ is NP-complete}$$

$$\text{CLIQUE} = \{(G, k) : G \text{ is an undirected graph with vertices V and edges E, } k \text{ is an integer, } 1 \leq k \leq |V|, \text{ and } G \text{ contains a } k\text{-clique}\} \text{ is NP-complete}$$

$$\text{3-SAT} = \{w : w \text{ is a Boolean wff, } w \text{ is in 3-CNF, and } w \text{ is satisfiable}\} \text{ is NP-complete}$$

3.38 INDEPENDENT-SET is NP-Complete

ER Chapter 28 Section 28.6.4

$$\text{INDEPENDENT-SET} = \{(G, k) : G \text{ is an undirected graph and } G \text{ contains an independent set of at least } k \text{ vertices}\} \text{ is NP-complete}$$

Let's prove this one.

Theorem 9. *INDEPENDENT-SET is* NP*-complete.*

Proof. Need to prove that the two properties hold

- Property 2. 3-SAT $\leq_P$ INDEPENDENT-SET
  - We have shown a mapping reduction R based on gadgets which runs in polynomial time and is correct.
- Property 1. INDEPENDENT-SET $\in$ NP
  - A certificate $\langle G, k, S \rangle$ can be verified in polynomial time as follows

    INDEPENDENT-SET-VERIFY($\langle G, k, S \rangle$)
    ```
    1  if |S| < k ∨ |S| > |V|
    2      return FALSE
    3  for v ∈ S
    4      for (u, v) ∈ E
    5          if u ∈ S
    6              return FALSE
    7  return TRUE
    ```
  - Clearly, $\text{timereq}(\text{INDEPENDENT-SET-VERIFY}) \in O(|S| \cdot |E| \cdot |S|)$
  - $|S|$ and $|E|$ are polynomial in size of G and k, hence INDEPENDENT-SET-VERIFY runs in polynomial time

□

3.39 TSP-DECIDE is NP-Complete

ER Chapter 28 Section 28.6.6

$$\begin{aligned}\text{TSP-DECIDE} = \{(G, c) : &\ G \text{ is an undirected graph with positive edge weights and} \\ &\ G \text{ contains a Hamiltonian circuit with cost} \leq c\} \\ &\ \text{is NP-complete}\end{aligned}$$

Let's prove this one too.

Theorem 10. *TSP-DECIDE is* NP*-complete.*

Proof. Need to prove that the two properties hold

- Property 1. TSP-DECIDE $\in$ NP
  - We have shown the polynomial-time, non-deterministic decider TSP-DECIDE
- Property 2. HAMILTONIAN-CIRCUIT $\leq_P$ TSP-DECIDE
  - Let $G = (V, E)$ be an unweighted, undirected graph
  - If $G \in$ HAMILTONIAN-CIRCUIT, it must contain exactly $|V|$ edges
  - So the mapping reduction R operates as follows:

    From G construct G', identical to G except that each edge has cost 1

    Return $(G', |V|)$
  - R runs in polynomial time
  - R is correct since G has a Hamiltonian circuit iff G' has one with cost $|V|$

□

3.40 NP-Complete Languages (continued)

ER Chapter 28 Section 28.6.1

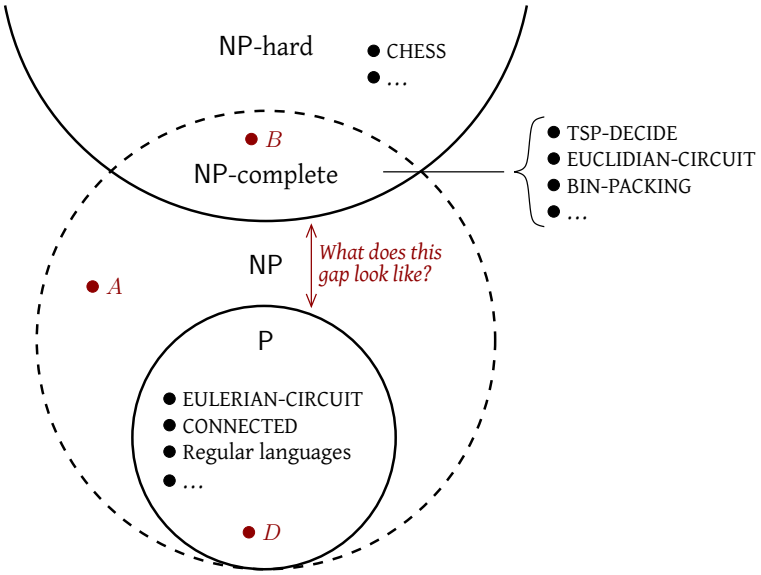
$$\begin{aligned}\text{SUBGRAPH-ISOMORPHISM} = \{(G_1, G_2) : &\ G_1 \text{ is isomorphic to some subgraph of } G_2\} \\ &\ \text{is NP-complete}\end{aligned}$$

$$\text{BIN-PACKING} = \{(S, c, k) : S \text{ is a set of objects each with associated size and set can be divided so that objects fit into } k \text{ bins, each of which has size } c\} \text{ is NP-complete}$$

$$\text{SHORTEST-SUPERSTRING} = \{(S, k) : S \text{ is a set of strings and } \exists \text{ some superstring } T \text{ s.t. every } s \in S \text{ is a substring of } T \text{ and } |T| \leq k\} \text{ is NP-complete}$$

3.41 Relation Between P, NP, NP-Complete and NP-Hard

ER Chapter 28 Section 28.7



If P = NP, then there is no "gap".

What happens if P ≠ NP?

3.42 Ladner's Theorem

ER Chapter 28 Section 28.7.1

Lemma 1. *Let B be any decidable language that is not in* P*. There exists a language $D \in$ P such that $A = D \cap B \neq \emptyset$, and the following holds:*

- $A \notin$ P *(what remains from the intersection, A, is intractable)*
- $A \leq_P B$ *(B is "at least as hard" as A)*
- $B \not\leq_P A$ *(B is "harder" than A)*

Proof. Omitted, see (Rich, 2008) or (Ladner, 1975). □

So, we have a way of making a language A that is not as hard as a given intractable language B, but is still not tractable.

Theorem 11. *If* P ≠ NP*, then there is something in the "gap", that is,*

$$\text{NP} \setminus (\text{P} \cup \text{NP-}\textit{complete}) \neq \emptyset$$

Proof. Relies on previous Lemma:

- Suppose that B is any NP-complete language
- If P ≠ NP, then B is not in P

- There exists $D \in \mathsf{P}$ from which we can compute $A = D \cap B$
- To check membership in A we must check membership in D *and* in B
- A must be in NP, since
  - membership in B can be verified in polynomial time
- So, using the Lemma, we have:
  - $A \notin \mathsf{P}$, but
  - It is not true that $B \leq_P A$
- Since B is in NP but is not deterministic, polynomial-time reducible to A, A is not NP-complete
- So, A is an example of an NP language that is neither in P nor NP-complete — thus, the "gap" is not empty

□

3.43 The Gap Between P and NP-Complete

Let's summarize here:

- There could be a "gap" between P and NP-complete
  - This gap would contain languages that are in NP, but not in P and not NP-complete
- Clearly, if $\mathsf{P} = \mathsf{NP}$, then there is no gap to talk about
- But if $\mathsf{P} \neq \mathsf{NP}$, then the gap is not empty (Ladner, 1975)
- Also, if the gap is not empty, then that proves that $\mathsf{P} \neq \mathsf{NP}$

Hence, we have that

Corollary 1. $\mathsf{P} \neq \mathsf{NP}$ *if and only if* $\mathsf{NP} \setminus (\mathsf{P} \cup \mathsf{NP}\text{-}complete) \neq \emptyset$.

3.44 Problems That Could Be in the Gap

There are many languages/problems that could be in the gap.

That is, languages $L \in \mathsf{NP}$ for which we cannot prove either of the following:

- $L \in \mathsf{P}$, or
- $L' \leq_P L$, where L' is NP-complete

Most problems that appear to be in the gap are not very "natural" though.

Typical example of "non-natural" problem in the gap:

$$\text{INTERSECTING-MONOTONE-SAT} = \{w : w \text{ is an intersecting monotone CNF formula and } w \text{ is satisfiable}\}$$

where

- Monotone CNF: every clause contains only positive literals or only negative literals
- Intersecting monotone CNF: every positive clause has some variable in common with every negative clause

Another one that we cannot prove to be in P nor to be NP-complete:

$$\text{SUM-ROOTS} = \{\langle (a_1, b_1), \ldots, (a_k, b_k) \rangle : (a_i, b_i) \in \mathbb{N}^2, \sum_{i=1}^{k} \sqrt{a_i} > \sum_{i=1}^{k} \sqrt{b_i}\}$$

> *"A major bottleneck in proving* NP*-completeness for geometric problems is a mismatch between the real-number and Turing machine models of computation: one is good for geometric algorithms but bad for reductions, and the other vice versa. Specifically, it is not known on Turing machines how to*

quickly compare a sum of distances (square roots of integers) with an integer or other similar sums, so even (decision versions of) easy problems such as the [Euclidian] minimum spanning tree are not known to be in NP." (Eppstein, Retrieved January 2019)

3.45 Small Differences Matter

ER Chapter 28 Section 28.7.1

Most "natural" problems in NP are either in P or are NP-complete.

It seems that natural problems in NP "snap to" being in P or NP-complete.

One candidate "natural" problem that we think might be in the gap is:

$$\text{GRAPH-ISOMORPHISM} = \{(G_1, G_2) : G_1 \text{ is isomorphic to } G_2\}$$

Recall that SUBGRAPH-ISOMORPHISM is NP-complete!

3.46 Two Similar Circuit Problems

ER Chapter 28 Section 28.7.2

$$\text{EULERIAN-CIRCUIT} = \{G : G \text{ is an undirected graph and } G \text{ contains a Eulerian circuit}\} \in \text{P}$$

$$\text{HAMILTONIAN-CIRCUIT} = \{G : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian circuit}\} \text{ is NP-complete}$$

3.47 Two Similar SAT Problems

ER Chapter 28 Section 28.7.3

$$\text{2-SAT} = \{w : w \text{ is a Boolean wff, w is in 2-CNF, and w is satisfiable}\} \in \text{P}$$

For example, $(\neg P \vee R) \wedge (S \vee \neg T)$ can be solved by Unit Propagation.

$$\text{3-SAT} = \{w : w \text{ is a Boolean wff, } w \text{ is in 3-CNF, and } w \text{ is satisfiable}\} \text{ is NP-complete}$$

For example, $(\neg P \vee R \vee T) \wedge (\neg S \vee \neg R \vee P) \wedge (S \vee \neg T \vee P)$ requires search.

3.48 Two Similar Path Problems

ER Chapter 28 Section 28.7.4

A simple path through a graph is a path with no repeated edges.

$$\text{SHORTEST-PATH} = \{(G, u, v, k) : G \text{ is an undirected graph, } u \text{ and } v \text{ are vertices in } G, k \geq 0 \text{ and there exists a simple path from } u \text{ to } v \text{ of length} \leq k\} \in \text{P}$$

$$\text{LONGEST-PATH} = \{(G, u, v, k) : G \text{ is an undirected graph, } u \text{ and } v \text{ are vertices in } G, k \geq 0 \text{ and there exists simple a path from } u \text{ to } v \text{ of length} \geq k\} \text{ is NP-complete}$$

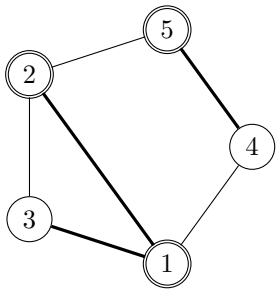
SHORTEST-PATH-DECIDE$(G = (V, E), u, v, k)$

```
1  M = {u}
2  for i = 1 to min(k, |E|)
3       for each n ∈ M
4            for each (n, m) ∈ E
5                 M = M ∪ {m}
6  if v ∈ M
7       return TRUE
8  return FALSE
```

SHORTEST-PATH-DECIDE runs in $O(|G|^3)$ time, hence SHORTEST-PATH $\in$ P

3.49 Two Similar Covering Problems

ER Chapter 28 Section 28.7.5



An edge cover C of a graph $G = (V, E)$ is a subset of E such that every $v \in V$ is an endpoint of one of the edges in C (see bold arrows in figure)

A vertex cover C of a graph $G = (V, E)$ is a subset of V such that every $(u, v) \in E$ touches one of the vertices in C (see marked vertices in figure).

EDGE-COVER $= \{(G, k) : G$ is an undirected graph and there exists an edge cover of G of size $\leq k\} \in$ P

VERTEX-COVER $= \{(G, k) : G$ is an undirected graph and there exists a vertex cover of G of size $\leq k\}$ is NP-complete

3.50 Two Similar Linear Programming Problems

ER Chapter 28 Section 28.7.7

LINEAR-PROGRAMMING $= \{Ax \leq B :$ there exists a rational vector $\mathbf{x}$ that satisfies all inequalities$\} \in$ P

INTEGER-LINEAR-PROGRAMMING $= \{Ax \leq B :$ there exists an integer vector $\mathbf{x}$ that satisfies all inequalities$\}$ is NP-complete

3.51 Diophantine Equations

ER Chapter 28 Section 28.7.8

A Diophantine equation is a polynomial equation in any number of variables with integer coefficients requiring integer solutions.

For example, if x, y, z, w are unknowns and a, b, n are constants:

$$ax + by = 1 \tag{1}$$
$$x^n + y^n = z^n \tag{2}$$
$$w^3 + x^3 = y^3 + z^3 \tag{3}$$
$$\frac{1}{x} + \frac{1}{y} + \frac{1}{z} = \frac{4}{n} \tag{4}$$

Equation (1) is a linear Diophantine equation.

Equation (2) has infinitely many solutions[5] for $n = 2$, and none for $n \geq 3$ (Diophantus, Fermat, and Bachet, 1670; Wiles, 1995).

Equation (3) has the solution $12^3 + 1^3 = 9^3 + 10^3 = 1729$ which was given by Ramanujan as an evident property of a taxicab number he had seen (1729).

Equation (4), which can be re-written as $4xyz = yzn + xzn + xyn$, was conjectured (Erdős, 1950) to have a positive integer solution for all $n \geq 2$.

Hilbert's tenth problem (Hilbert et al., 1902) asks

> *"[…] to devise a process according to which it can be determined in a finite number of operations whether the [Diophantine] equation is solvable in rational integers."*

We could re-state it as follows

$$\text{TENTH} = \{w : w \text{ is a system of Diophantine equations that has an integer solution}\}$$

This problem was proved by Matiyasevich (1970) to be undecidable[6].

Variants of TENTH, however, are decidable, and they snap nicely around the "gap":

$$\text{TENTH}' = \{w : w \text{ is a system of linear Diophantine equations or in the form } ax^k = c \text{ that has an integer solution}\} \in \mathsf{P}$$

$$\text{TENTH}'' = \{w : w \text{ is a system of Diophantine equations in the form } ax^2 + by = c \text{ that has an integer solution}\} \text{ is NP-complete}$$

Example in TENTH′:

> *A farmer buys 100 animals for $100.00. The animals include at least one cow, one pig, and one chicken, but no other kind. If a cow costs $10.00, a pig costs $3.00, and a chicken costs $0.50, how many of each did he buy?*

$$\begin{aligned} 10x_{\text{cows}} + 3x_{\text{pigs}} + \frac{1}{2}x_{\text{chickens}} &= 100 \\ x_{\text{cows}} + x_{\text{pigs}} + x_{\text{chickens}} &= 100 \end{aligned}$$

Easy to show that he bought 9 cows, 3 pigs, 2 chickens (write a deterministic algorithm to solve this general problem and show that it is polynomial).

[5] Proved by the Greeks (Heath and Euclid, 1956), possibly known since the Babylonians (Robson, 2001).

[6] That this problem was solved by showing that there cannot be any such algorithm contradicted Hilbert's philosophy of mathematics.

3.52 Decision Problems vs. Search Problems

So far, we have focused on decision problem variants of search problems.

What about the corresponding search questions?

- Given $G = (V, E)$, compute a Eulerian cycle
- Given $G = (V, E)$, compute a Hamiltonian cycle
- Given $G = (V, E)$, compute an independent set of size k
- Given a 3-CNF formula, compute a satisfying assignment

3.53 Search by Solving Decision Problems

Can we use a procedure for deciding membership to actually find a certificate?

In other words, can we exploit an decision oracle for search?

We can show this for SAT very easily:

Theorem 12. *The SAT search problem is solvable in polynomial time given a polynomial-time verifier (oracle)* SAT-DECIDE *for the SAT decision problem.*

Proof. We can use SAT-DECIDE(ϕ) as follows to compute a satisfying assignment for a given formula ϕ:

SAT-SEARCH$(\phi(x_1, \ldots, x_n))$
1 **if** $\neg$SAT-DECIDE(ϕ)
2 **return** FALSE
3 **for** $i = 1$ to n
4 **if** SAT-DECIDE$(\phi(x_1 = b_1, \ldots, x_{i-1} = b_{i-1}, \top, x_{i+1}, \ldots, x_n))$
5 $b_i = \top$
6 **else** $b_i = \bot$
7 **return** $\langle b_1, \ldots, b_n \rangle$

The deterministic procedure SAT-VERIFY makes at most n calls to the oracle (polynomial-time procedure) SAT-VERIFY. □

This property of SAT is called self-reducibility, that is, checking the satisfiability of a formula on n variables reduces to checking the satisfiability of two formulas on $n - 1$ variables.

The same holds for other problems in NP, for instance:

Theorem 13. *The CLIQUE search problem is solvable in polynomial time given a polynomial-time verifier (oracle)* CLIQUE-DECIDE *for the CLIQUE decision problem.*

Proof. For any graph $G = (V, E)$ and any $v \in V$, let $G \setminus v$ be the graph G after removing from it the node v and all edges adjacent to it. We are given CLIQUE-DECIDE(G, k) which decides if $G = (V, E)$ has a clique of size k. We can use it to find a subset of V that is a clique of size k if one exists as follows:

CLIQUE-SEARCH(G, k)
1 **if** $\neg$CLIQUE-DECIDE(G, k)
2 **return** FALSE
3 **for** $v \in V$
4 **if** CLIQUE-DECIDE$(G \setminus v, k)$
5 $G = G \setminus v$
6 **return** an arbitrary subset of k nodes of G

The procedure is correct because

- At the first iteration of the for loop, we know that G has a clique of size k

- This remains true for every iteration
- Also, if v is not removed, then all cliques of size k contain v
- At the last iteration, all remaining vertices in G are members of all cliques of size $\geq k$
- As there could be larger cliques, we randomly select k vertices to return

The deterministic procedure CLIQUE-VERIFY makes at most $|V|$ calls to the oracle (polynomial-time procedure) CLIQUE-VERIFY. □

Does this work with all problems in NP? Suppose $L \in$ NP and we have polynomial-time verifier V for it:

- Since $L \in$ NP, then $L \leq_P$ SAT
- But V only works for L, it is not an oracle for SAT
- So we can't run the SAT self-reducibility algorithm to find $x \in L$

3.54 Decision vs. Search for NP-complete Problems

But if L is NP-complete, then that's another story, because of the following result:

Theorem 14. *Let L-DECIDE be a polynomial-time verifier (oracle) for a language L. Then there exist polynomial-time computable functions f and g such that*

- *$x \in L$ iff $f(x) \in$ SAT*
- *If A is a satisfying assignment to $f(x)$, then L-DECIDE will accept certificate $\langle x, g(x, A)\rangle$*

Proof. Based on how the reduction of the Cook-Levin Theorem works. □

This tells us that we can indeed use verification to solve search problems, as long as they are NP-complete:

Theorem 15. *Given the decision problem associated to an NP-complete language L, the corresponding search problem is solvable in polynomial time given a polynomial-time verifier (oracle) L-DECIDE for the decision problem.*

Proof. We are given x and an oracle L-DECIDE for L. We want to find a certificate $\langle x, w\rangle$ if one exists.

- We would like to ask the oracle questions, but what should we ask? We don't know the structure of the problem…
- But we do know (theorem above) that there is a function f that transforms x into a formula $f(x)$ and that $x \in L$ iff $f(x) \in$ SAT.
- But how do we find an assignment A for $f(x)$ without the oracle for SAT?
- We can exploit the fact that SAT $\leq_P L$ (since L is NP-complete), hence there is a reduction R_L that reduces SAT to L
- We can build an oracle for SAT by using this reduction and L-DECIDE:

```
SAT-DECIDE(φ)
1  α = R_L(φ)
2  return L-DECIDE(α)
```

- So now we can use the self-reducibility property of SAT to find a satisfying assignment A for $f(x)$
- Again thanks to the theorem, we can build a certificate $\langle x, g(x, A)\rangle$ for our original problem $x \in L$

□

Part IV

Other Time Complexity Classes

4.1 The Class coNP

ER Chapter 28 Section 28.8

Remember we said that P was closed under complement?

We do not know if the same holds for NP. Let us define the following:

$L \in$ coNP iff $\neg L \in$ NP

For example, TSP-DECIDE $\in$ NP, while NOT-TSP-DECIDE $\in$ coNP

$$\text{NOT-TSP-DECIDE} = \{(G, c) : G \text{ is an undirected graph with positive edge weights and } G \text{ does not contain a Hamiltonian circuit with cost} \leq c\}$$

What does the class coNP actually mean?

4.2 coNP and NDTMs

How to characterize coNP in terms of a NDTM?

By definition, $L \in$ coNP iff $\neg L \in$ NP. Hence there exists a NDTM M such that

- If $x \notin \neg L$, then $M(x)$ rejects for all computation paths
- If $x \in \neg L$, then $M(x)$ accepts for some computation path

Hence, we can construct a NDTM M' which accepts (rejects) iff M rejects (accepts):

- If $x \notin \neg L$ (iff $x \in L$), then $M'(x)$ accepts for all computation paths
- If $x \in \neg L$ (iff $x \notin L$), then $M'(x)$ rejects for some computation path

Overall,

- NP is the class of problems for which a qualifying certificate can be checked efficiently
  - Because there is a verifying TM that runs in polynomial time
- NP is the class of problems that have succinct qualifying certificates
  - This certificate is an accepting path of a NDTM, which can only be polynomial in size because it took one computation path polynomial time to write it
- coNP is the class of problems for which a disqualifying certificate can be checked efficiently
  - Because there is a verifying TM that runs in polynomial time
- coNP is the class of problems that have succinct disqualifying certificates
  - This certificate is a rejecting path of a NDTM, which can only be polynomial in size because it took one computation path polynomial time to write it

4.3 Relating NP and coNP

ER Chapter 28 Section 28.8

The class coNP helps to understand the relation between P and NP.

Theorem 16. *If* NP $\neq$ coNP *then* P $\neq$ NP.

Proof. By contradiction, assume that P $=$ NP

- But we know that P is closed under complement
- Hence, P $=$ NP $=$ coNP, which invalidates the theorem's hypothesis
- Therefore, if NP is not closed under complement, then NP cannot be equal to P

□

It would be "nice" if we could prove that NP is not closed under complement, because it would prove that P $\neq$ NP.

But proving the opposite, that is, NP $=$ coNP, does not imply that P $=$ NP.

That is, it is possible that NP $=$ coNP but that that class is nevertheless larger than P.

In fact, we can characterize what NP $=$ coNP would mean a bit more precisely:

Theorem 17. NP $=$ coNP *iff* $\exists\, L$ *such that* L *is* NP*-complete and* $\neg L \in$ NP*.*

Proof. See (Rich, 2008) if interested. □

This is somewhat intuitive: if the complement of some NP-complete problem (a problem that "can be used to solve" all problems in NP) remains in NP, then NP is closed under complement.

4.4 Relating P, NP and coNP

Theorem 18. *If* $L \in$ P *then* $L \in$ NP *and* $L \in$ coNP*.*

Proof. We know that $L \in$ NP because P $\subseteq$ NP. Hence, by definition $\neg L \in$ coNP. Since P is closed under complement, we know that $\neg L \in$ P, and therefore $\neg L \in$ NP. Therefore, $\neg\neg L = L \in$ coNP. □

That is,

$$\text{P} \subseteq \text{NP}$$
$$\text{P} \subseteq \text{coNP}$$

4.5 coNP-Complete Languages

A language L might have these properties:

- Property 1. $L \in$ coNP
- Property 2. $L' \leq_P L$ for all $L' \in$ coNP

L is coNP-hard iff it possesses Property 2.

L is coNP-complete iff it possesses both Property 1 and Property 2.

A coNP-hard language is at least as hard as any other language in coNP.

All coNP-complete languages can be viewed as being equivalently hard.

However, we don't need a "seed" coNP-complete language (or, finding a "seed" language does not require a complex proof), because:

Theorem 19. L *is* NP*-complete iff* $\neg L$ *is* coNP*-complete.*

Proof. We prove the $\Rightarrow$ direction (the opposite is symmetric):

- We know that L is NP-complete
- We need to prove that any $\neg L' \in$ coNP can be reduced to $\neg L$
  - By definition of the class coNP, we have that $\neg\neg L' = L' \in$ NP
  - Since L is NP-complete, there exists a poly-time reduction R from L' to L
  - So, $x \in L'$ iff $R(x) \in L$
  - So, $x \in \neg L'$ iff $R(x) \in \neg L$
  - Hence, R is a reduction from $\neg L'$ to $\neg L$
  - Hence, $\neg L$ is coNP-complete

□

Examples of coNP-complete problems: complement of all problems we have shown to be NP-complete!

4.6 VALIDITY is coNP-Complete

A wff is valid iff it is true for all assignments of values to variables.

$$\text{VALIDITY} = \{w : w \text{ is a Boolean wff and } w \text{ is valid}\} \text{ is coNP-complete}$$

In fact,

- w is valid iff $\neg w$ is unsatisfiable, that is $\neg w \in \neg\text{SAT}$
- The language $\neg$SAT is coNP-complete because SAT is NP-complete
- Hence, any problem in coNP can be reduced to $\neg$SAT
- Hence, any problem in coNP can be reduced to VALIDITY

4.7 Possible Relations Between P, NP and coNP

Three possibilities:

- $\text{P} = \text{NP} = \text{coNP}$
- $\text{NP} = \text{coNP}$ but $\text{P} \neq \text{NP}$
- $\text{NP} \neq \text{coNP}$ and $\text{P} \neq \text{NP}$ (this is the current consensus)

Problems/languages that have both short qualifying certificates and short disqualifications belong to both NP and coNP.

Let $\text{DP} = \text{NP} \cap \text{coNP}$ (Difference Polynomial Time).

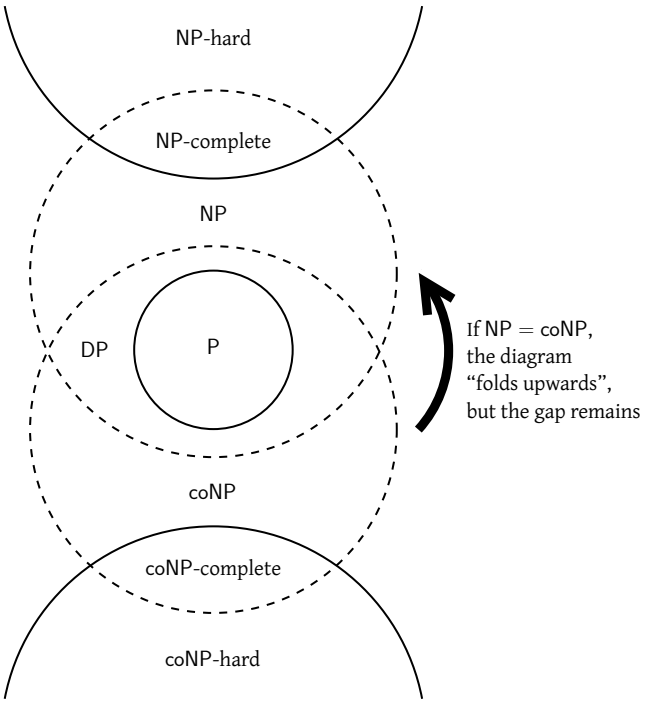
Note that $\text{P} \subseteq \text{DP}$, because $\text{P} \subseteq \text{NP}$ and $\text{P} \subseteq \text{coNP}$.

We do not know if $\text{P} = \text{DP}$, that is, if there is a problem with short certificates and short disqualifiers that is not tractable.

Example of problem in DP (Pratt, 1975):

$$\text{PRIMES} = \{n : n \in \mathbb{N} \text{ and } n \text{ is prime}\} \in \text{DP}$$

It turns out that PRIMES $\in$ P, because an efficient algorithm for primality testing was discovered relatively recently (Agrawal, Kayal, and Saxena, 2004).



4.8 Beyond NP: The Class EXP

ER Chapter 28 Section 28.9

$L \in$ EXP iff

- there is some TM M that decides L, and
- $\text{timereq}(M) \in O(2^{(n^k)})$ for some positive integer k.

For example,

$$\text{CHESS} = \{b : b \text{ is a configuration of an } n \times n \text{ chess board and there is a guaranteed win for the current player}\} \in \text{EXP}$$

We will see later how the class EXP relates to classes P and NP.

4.9 EXP-Completeness

ER Chapter 28 Section 28.9

A language L might have these properties:

- Property 1. $L \in$ EXP
- Property 2. $L' \leq_P L$ for all $L' \in$ EXP

L is EXP-hard iff it possesses Property 2.

L is EXP-complete iff it possesses both Property 1 and Property 2.

An EXP-hard language is at least as hard as any other language in EXP.

All EXP-complete languages can be viewed as being equivalently hard.

It turns out that CHESS is EXP-complete if, as we scale n, we also add pieces.

Part V

The Language Class PSPACE

5.1 Space Requirement

ER Chapter 29 Section 29.1

If M is a TM that halts on all inputs, then

$$\text{spacereq}(M) = f(n) = \text{max. number of tape squares read on any input of length } n$$

If M is a NDTM all of whose computational paths halt on all inputs, then

$$\text{spacereq}(M) = f(n) = \text{max. number of tape squares read on any path executed on any input of length } n$$

5.2 Example: CONNECTED

ER Chapter 29 Section 29.1.1

$$\text{CONNECTED} = \{G = (V, E) : G \text{ is an undirected graph and } G \text{ is connected}\}$$

CONNECTED$(G = (V, E))$

```
1  Set all vertices to be unmarked
2  Select a vertex v
3  L = {v}
4  n_marked = 1
5  while L ≠ ∅
6       v = POP(L)
7       for (v, u) ∈ E
8            if u not marked
9                 Mark u
10                L = L ∪ {u}
11                n_marked = n_marked + 1
12 if n_marked = |V|
13      return TRUE
14 return FALSE
```

CONNECTED$(G = (V, E))$ uses space for:

- Storing the marks on the vertices
- The list L of marked vertices whose successors have not yet been examined
- The counter n_{marked}, which can be stored in binary in $\log(|V|)$ bits

So, $\text{spacereq}(\text{CONNECTED}(G)) \in O(|G|)$.

5.3 Example: SAT

ER Chapter 29 Section 29.1.1

$$\text{SAT} = \{w : w \text{ is a Boolean wff and } w \text{ is satisfiable}\}$$

We already have a non-deterministic procedure for deciding SAT:

SAT-DECIDE(w)

```
1  for each variable v in w
2       CHOOSE({⊤, ⊥}) and assign it to v
3  if EVAL(w)
4       return TRUE
5  return FALSE
```

Clearly, $\text{spacereq}(\text{SAT-DECIDE}(w)) \in O(w)$.

How about a deterministic procedure?

SAT-DECIDE-DETERMINISTIC(w)

```
1  for each binary string b_1 b_2 ... b_#variables
2      Assign value b_i to variable x_i in w
3      if EVAL(w)
4          return TRUE
5  return FALSE
```

Analysis of SAT-DECIDE-DETERMINISTIC(w):

- Each iteration of the for loop requires maintaining one string of length #variables
- All other lines have constant space requirement too

So, $\text{spacereq}(\text{SAT-DECIDE-DETERMINISTIC}(w)) \in O(w)$ as well!

Note: we do not need to create a truth table, which would have size $2^{\#\text{variables}}$

5.4 Relating Time and Space Complexity

ER Chapter 29 Section 29.1.2

Theorem 20. *Given a TM M, and assuming that $\text{spacereq}(M) \geq n$, the following holds:*

$$\text{spacereq}(M) \leq \text{timereq}(M) \in O(c^{\text{spacereq}(M)})$$

Proof. Proving that $\text{spacereq}(M) \leq \text{timereq}(M)$:

- $\text{spacereq}(M)$ is bounded by $\text{timereq}(M)$ since M must use at least one time step for every tape square it visits.

Proving that $\text{timereq}(M) \in O(c^{\text{spacereq}(M)})$:

- Since M halts, the number of steps it can execute is bounded by the number of distinct configurations that it can enter
  - So $\text{timereq}(M) \leq \text{MaxConfigs}(M)$
- Let K be M's set of states and Γ be its tape alphabet
  - Note that Γ contains (at least) input alphabet Σ and blank symbol "$\square$"
- Then, $\text{MaxConfigs}(M) = |K| \cdot |\Gamma|^{\text{spacereq}(M)} \cdot \text{spacereq}(M)$
- If $|\Gamma| \leq c$ where c is some constant, then $\text{MaxConfigs}(M) \in O(c^{\text{spacereq}(M)})$
- Therefore, $\text{timereq}(M) \in O(c^{\text{spacereq}(M)})$

□

5.5 The Language Classes PSPACE and NPSPACE

ER Chapter 29 Section 29.2

$L \in$ PSPACE iff

- there is some TM M that decides L, and
- $\text{spacereq}(M) \in O(n^k)$ for some constant k.

$L \in$ NPSPACE iff

- there is some NDTM M that decides L, and
- $\text{spacereq}(M) \in O(n^k)$ for some constant k.

5.6 Relation Between PSPACE and NPSPACE

ER Chapter 29 Section 29.2

It turns out that PSPACE $=$ NPSPACE.

To see why, we need a result obtained by Savitch (1970):

Theorem 21. *If L can be decided by a NDTM M and $\mathrm{spacereq}(M) \geq n$, then there is a TM M' that also decides L and $\mathrm{spacereq}(M') \in O(\mathrm{spacereq}(M)^2)$.*

Proof. Omitted, see (Rich, 2008). □

This allows to show that

Theorem 22. PSPACE $=$ NPSPACE.

Proof. We should prove that

- If $L \in$ PSPACE then $L \in$ NPSPACE, but of course this is trivial
- If $L \in$ NPSPACE then $L \in$ PSPACE
  - If $L \in$ NPSPACE then there is some NDTM M that decides it and $\mathrm{spacereq}(M) \in O(n^k)$ for some k
  - Savitch tells us[7] that there is a TM M' that decides L it and $\mathrm{spacereq}(M') \in O(\mathrm{spacereq}(M)^2 = n^{2k})$
  - Hence, $L \in$ PSPACE as well

□

5.7 Relation Between P, NP, PSPACE and EXP

ER Chapter 29 Section 29.2

Theorem 23. P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXP.

Proof. We have already shown that P $\subseteq$ NP. To show that NP $\subseteq$ PSPACE:

- If $L \in$ NP, then it is decided by some NDTM M in polynomial time
- In polynomial time, M cannot use more than polynomial space since it takes a least one time step to visit a tape square
- This means that $L \in$ NPSPACE
- Since NPSPACE $=$ PSPACE (Savitch), then $L \in$ PSPACE

To show that PSPACE $\subseteq$ EXP:

- If $L \in$ PSPACE, then it is decided by some TM M in polynomial space
- We have shown that $\mathrm{spacereq}(M) \leq \mathrm{timereq}(M) \in O(c^{\mathrm{spacereq}(M)})$
- Hence, $L \in$ EXP

□

5.8 PSPACE-Completeness

ER Chapter 29 Section 29.3

A language L might have these properties:

- Property 1. $L \in$ PSPACE
- Property 2. $L' \leq_P L$ for all $L' \in$ PSPACE

L is PSPACE-hard iff it possesses Property 2.

L is PSPACE-complete iff it possesses both Property 1 and Property 2.

A PSPACE-hard language is at least as hard as any other language in PSPACE.

All PSPACE-complete languages can be viewed as being equivalently hard.

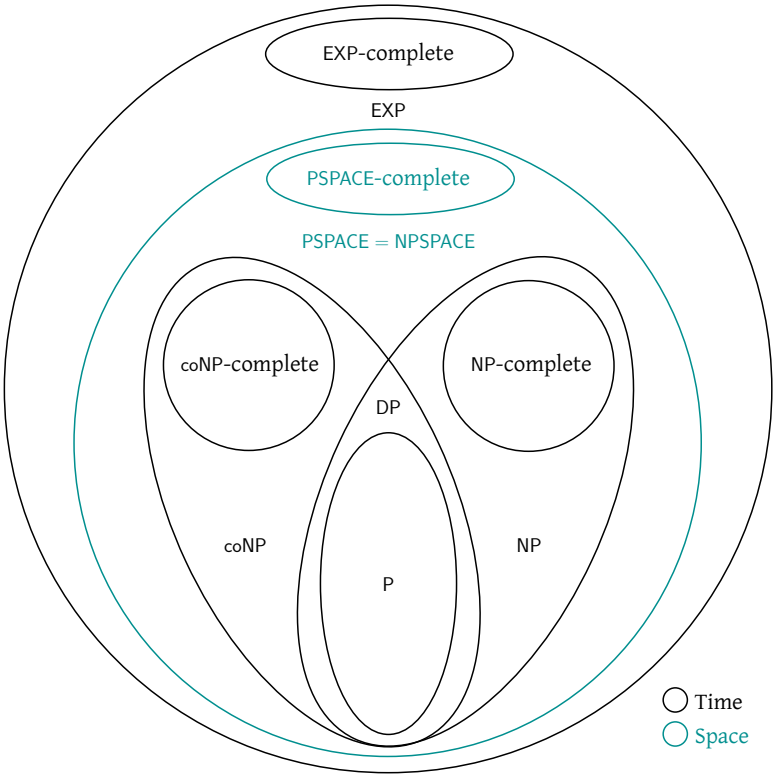
[7] Assuming $k \geq 1$; the proof actually includes the case $k < 1$, but we omit it here.

5.9 PSPACE-Completeness, P, and NP

ER Chapter 29 Section 29.3

If any PSPACE-complete language is also in NP, then all of them are and NP = PSPACE.

If any PSPACE-complete language is also in P, then all of them are and P = NP = PSPACE.



5.10 A First PSPACE-Complete Language

ER Chapter 29 Section 29.3.1

SAT won't work because it is NP-complete and we suspect that there are PSPACE languages that are not in NP.

A quantified Boolean formula (QBF) is a Boolean formula where each variable x_i may be bound by a quantifier $Q_i \in \{\forall, \exists\}$.

If all variables are bound by a quantifier, the formula is closed or fully quantified.

$$w = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n . \phi(x_1, x_2, \ldots, x_n)$$

where ϕ is a Boolean formula.

$$\text{TQBF} = \{w : w \text{ is a true quantified Boolean formula}\}$$

$$w_1 = \forall X \exists Y . ((X \vee Y) \wedge (\neg X \vee \neg Y)) \in \text{TQBF}$$
$$w_2 = \forall X \forall Y . ((X \vee Y) \wedge (\neg X \vee \neg Y)) \notin \text{TQBF}$$

Can we state a deterministic algorithm for TQBF?

TQBF-DECIDE($w = Q_1 x_1 \ldots Q_n x_n . \phi(x_1, \ldots, x_n)$)

```
1  if w contains no quantifiers
2      return EVAL(φ)
3  A = TQBF-DECIDE(Q_2 x_2 ... Q_n x_n . φ(⊥, x_2, ..., x_n))
4  B = TQBF-DECIDE(Q_2 x_2 ... Q_n x_n . φ(⊤, x_2, ..., x_n))
5  if Q_1 = ∃
6      return EVAL(A ∨ B)
7  return EVAL(A ∧ B)
```

Analysis of TQBF-DECIDE(w):

- For each quantifier, the algorithm makes two recursive calls on a smaller sub-problem
- But the sub-problem is only linearly smaller, hence $\text{timereq}(\text{TQBF-DECIDE}(w)) \in O(2^n)$
- For each recursive invocation, is should store result of computing A and B
- Depth of recursion is n, hence $\text{spacereq}(\text{TQBF-DECIDE}(w)) \in O(n)$

5.11 TQBF is PSPACE-Complete

ER Chapter 29 Section 29.3.2

Our "first" PSPACE-complete problem is indeed TQBF.

Theorem 24. *TQBF is* PSPACE*-complete.*

Proof. We've done half of it already (albeit the easy half):

- TQBF $\in$ PSPACE because we have shown a linear-space, deterministic procedure to decide it

What remains is to prove that TQBF is PSPACE-hard:

- Done by constructing a polynomial-time reduction to TQBF from any $L \in$ PSPACE, similar to Cook-Levin theorem for proving that SAT is NP-complete — see (Rich, 2008) if interested

□

5.12 The Essence of PSPACE

ER Chapter 29 Section 29.3.3

A certificate of membership for an NP-complete problem is one that is short (polynomial).

A certificate of membership for a PSPACE-complete problem is a winning strategy for a two-player game with perfect information.

For example, CHESS, where players make alternate moves.

What is a winning strategy for the first player?

P_1 has a winning strategy iff
  $\exists$ a 1st first move for P_1 such that
    $\forall$ possible 1st moves of P_2
      $\exists$ a 2nd move P_1 such that
        $\forall$ possible 2nd moves of P_2
          ...
            P_1 wins at the end

The problem of deciding whether P_1 has a winning strategy seems to require searching the tree of all possible moves

- If the length of a game is bounded by some polynomial function of the size of the game, then the game is likely to be PSPACE-complete
- If the length of the game grows exponentially with the size of the game, then the game is likely not be solvable in polynomial space
  - But it is likely to be solvable in exponential time and thus to be EXP-complete.

A number of complexity results have been proven (Eppstein, Retrieved August 2020) for games and puzzles. In essence:

- The absence of a "general-purpose trick" often leads a puzzle to be NP-hard
- The tree of potential interactions in a game typically leads to PSPACE-hardness

5.13 Languages and Automata

ER Chapter 29 Section 29.3.3

$$\text{NeqNDFSMs} = \{(M_1, M_2) : M_1 \text{ and } M_2 \text{ are non-deterministic FSMs and } L(M_1) \leq L(M_2)\} \text{ is PSPACE-complete}$$

$$\text{NeqREGEX} = \{(E_1, E_2) : E_1 \text{ and } E_2 \text{ are regular expressions and } L(E_1) \leq L(E_2)\} \text{ is PSPACE-complete}$$

$$\text{2FSMs-INTERSECT} = \{(M_1, M_2) : M_1 \text{ and } M_2 \text{ are deterministic FSMs and } L(M_1) \cap L(M_2) \neq \emptyset\} \in \text{P}$$

$$\text{FSMs-INTERSECT} = \{(M_1, M_2, \ldots, M_n) : M_i \text{ are deterministic FSMs and } \exists \text{ some string accepted by all of them}\} \text{ is PSPACE-complete}$$

$$\text{CONTEXT-SENSITIVE-MEMBERSHIP} = \{(G, w) : x \in L(G)\} \text{ is PSPACE-complete}$$

Part VI

Overview of Complexity Classes

6.1 What We Know So Far

So far, we have shown that

$$\text{PSPACE} = \text{NPSPACE}$$
$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXP}$$

We think that all of the inclusions are strict, but we can prove this in one case only…

6.2 Time Constructible Functions

ER Chapter 28 Section 28.9.1

Given $n \in \mathbb{N}$, its unary representation is

$$1^n = \underbrace{11\ldots1}_{n \text{ times}}$$

A function $t : \mathbb{N} \mapsto \mathbb{N}$ is time-constructible iff

- $t(n) \in \Omega(n \log n)$, and
- There is a TM M that maps 1^n to the binary representation of $t(n)$ and $\text{timereq}(M) \in O(t(n))$

So, the question is: given $t : \mathbb{N} \mapsto \mathbb{N}, t(n) \geq n \log n$, and given an input n in unary, how long does it take to compute the value $t(n)$ in binary? If the answer is $O(t(n))$, then the function is time-constructible.

In essence, a function $t(n)$ is not time-constructible if you cannot read the input 1^n in time that is less than $t(n)$.

So, time-constructible functions are functions that can serve as upper bounds for TM computations.

For example,

- $t(n) = c$ is not time-constructible, because $c \notin \Omega(n \log n)$
- $t(n) = cn$ is not time-constructible, because $cn \notin \Omega(n \log n)$
- $t(n) = 2^n$ is time-constructible, because
  - $2^n \in \Omega(n \log n)$, and we can construct a TM M that does the following:
    Write as many 0's as there are 1's in the unary representation of n, each time advancing the head by one, requiring time $O(n)$
    Write a 1 on the tape in the head's current position, requiring time $O(1)$
    The result of 2^n is now represented in binary (LSB-first) on the tape

All polynomial functions in $\Omega(n \log n)$ are time-constructible.

The functions $n \log n, n\sqrt{n},$ and $n!$ are also time-constructible.

6.3 Deterministic Time Hierarchy Theorem

ER Chapter 28 Section 28.9.1

Theorem 25 (Deterministic Time Hierarchy Theorem)**.** *For any time-constructible function $t(n)$, there exists a language $L_{t(n)}$ that is deterministically decidable in $O(t(n))$ time but that is not deterministically decidable in $o\left(\frac{t(n)}{\log t(n)}\right)$ time.*

Proof. Omitted, see (Rich, 2008) if interested. □

This means, for instance, that

- There are problems that are solvable in time n^2 but not time n, because $n \in o\left(\frac{n^2}{\log n^2}\right)$
- There are problems that are solvable in time $2^{(n^k)}$ but not time n^k, because

$$n^k \in o\left(\frac{2^{(n^k)}}{\log 2^{(n^k)}}\right) = o\left(\frac{2^{(n^k)}}{n^k}\right)$$

Note:

- We are not saying that if L is deterministically decidable in $O(2^{(n^k)})$ then it is not deterministically decidable in $O(n^k)$
  - It's easy to make a very inefficient algorithm that requires exponential time to solve a simple problem!
- We are saying that there exists a language L that is deterministically decidable in exponential time but not in polynomial time

That's all we need to prove that

Corollary 2. P $\subset$ EXP.

6.4 Provably Intractable Problems

ER Chapter 28 Section 28.9.2

Since P $\subset$ EXP, we know that there are decidable problems for which no efficient algorithm exists.

Most importantly, this is true for every EXP-complete problem, because

- These are problems that are at least as hard as every other problem in EXP
- Every other problem in EXP obviously includes some problem that is not deterministically decidable in polynomial time (which we know exists thanks to the previous theorem)
- This is the reason we use the notion of completeness

So, CHESS is provably intractable, in the sense it is impossible to come up with deterministic polynomial-time algorithm for it.

6.5 A Glimpse of the Wider Complexity Landscape

There are many other interesting complexity classes beyond the ones we have shown[8] (see also summary figure in Section 6.7):

Class	*Computational model*	*Time/Space requirement*
L	TM	$\text{spacereq}(M) \in O(\log n)$
NL	NDTM	$\text{spacereq}(M) \in O(\log n)$
P	TM	$\text{timereq}(M) \in O(n^k)$
NP	NDTM	$\text{timereq}(M) \in O(n^k)$
PSPACE	TM	$\text{spacereq}(M) \in O(n^k)$
EXP	TM	$\text{timereq}(M) \in O(2^{(n^k)})$
NEXP	NDTM	$\text{timereq}(M) \in O(2^{(n^k)})$
EXPSPACE	TM	$\text{spacereq}(M) \in O(2^{(n^k)})$

Examples of problems in these new classes:

$$\text{MAJORITY} = \{x : x \text{ is a binary string and } x \text{ has at least as many 1's as 0's}\} \in \mathsf{L}$$

$$\text{PATH} = \{(G, u, v) : G \text{ is a directed graph with a path from } u \text{ to } v\} \text{ is NL-complete}$$

$$\text{2-SAT} = \{w : w \text{ is a Boolean wff, } w \text{ is in 2-CNF, and } w \text{ is satisfiable}\} \text{ is NL-complete}$$

[8] For an even wider landscape, take a look at University of Waterloo's "Complexity Zoo" at `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`.

Overview of relations between complexity classes:

$$\begin{aligned} \mathsf{PSPACE} &= \mathsf{NPSPACE} \\ \mathsf{EXPSPACE} &= \mathsf{NEXPSPACE} \\ \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} &\subseteq \mathsf{EXP} \subseteq \mathsf{NEXP} \subseteq \mathsf{EXPSPACE} \\ \mathsf{NL} &\subset \mathsf{PSPACE} \subset \mathsf{EXPSPACE} \\ \mathsf{P} &\subset \mathsf{EXP} \end{aligned}$$

We think that all of the inclusions are strict.

Note that polynomials are closed under squaring, but $O(\log n)$ is not, which is why Savitch's theorem cannot tell us the relationship between L and NL.

Important unknown relations:

- $\mathsf{L} \stackrel{?}{=} \mathsf{NL}$
- $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$
- $\mathsf{NP} \stackrel{?}{=} \mathsf{PSPACE}$
- $\mathsf{PSPACE} \stackrel{?}{=} \mathsf{EXP}$
- $\mathsf{EXP} \stackrel{?}{=} \mathsf{NEXP}$
- $\mathsf{NEXP} \stackrel{?}{=} \mathsf{EXPSPACE}$

That is,

- we don't know how to prove that non-determinism makes a difference
- we don't know how to prove that space is more powerful than time

Important known relations:

- $\mathsf{P} \neq \mathsf{EXP}$ (Deterministic Time Hierarchy Theorem)
- $\mathsf{NP} \neq \mathsf{NEXP}$ (Non-deterministic Time Hierarchy Theorem)
- $\mathsf{PSPACE} \neq \mathsf{EXPSPACE}$ (Space Hierarchy Theorem)

That is, we can prove that exponential gaps make a difference (when measuring the same resource bound).

However, the Hierarchy Theorems provide no means to relate deterministic and non-deterministic complexity, or time and space complexity.

6.6 The Class NEXP and Succinct Representation

The class NEXP is interesting in that it captures the difficulty of succinct variants of problems in NP.

A succinct variant of a problem is one in which the input can be represented succinctly thanks to some special structure.

For example, instead of providing a graph $G = (V, E)$ as input (hence, input size measured in terms of $|V|$), we input:

- The number of vertices of the graph, represented in binary
- Some compact rule to determine if two nodes are connected

This is something one might actually do if their graph is huge, in which case it makes practical sense to study the complexity of the succinct problem.

One such representation is the Small Circuit Representation (SCR) of a graph, defined as follows:

- Let $G = (V, E)$ be a graph with verices $V = \{v_1, \ldots v_m\}$ where $m \leq 2^n$
- Let the binary representation of the index of vertex v_i be an n-bit string $i_{(2)}$
- C_G is a SCR of G if:
  - C_G is a combinatorial circuit[9]

[9]The output is a pure function of the current input only. This is in contrast to a sequential circuit, where the output depends also on the history of the input, i.e., sequential circuits have memory, while combinational ones don't.

- C_G has two inputs of n bits each
- C_G has $r \in O(n^k)$ gates with k constant
- The output of C_G is given by

$$C_G(i_{(2)}, j_{(2)}) = \begin{cases} ? & \text{if } v_i \notin V \vee v_j \notin V \\ 0 & \text{if } (v_i, v_j) \notin E \\ 1 & \text{if } (v_i, v_j) \in E \end{cases}$$

Many graph properties become harder to decide in the succinct variant of the problem compared to the natural formulation (Galperin and Wigderson, 1983).

Some concrete examples (Yannakakis and M., 1986):

$$\text{HAMILTONIAN-CIRCUIT} = \{G : G \text{ is an undirected graph and } G \text{ contains a Hamiltonian circuit}\} \text{ is NP-complete}$$

$$\text{HAMILTONIAN-CIRCUIT-SUCCINCT} = \{C_G : C_G \text{ is a SCR of an undirected graph } G \text{ and } G \text{ contains a Hamiltonian circuit}\} \text{ is NEXP-complete}$$

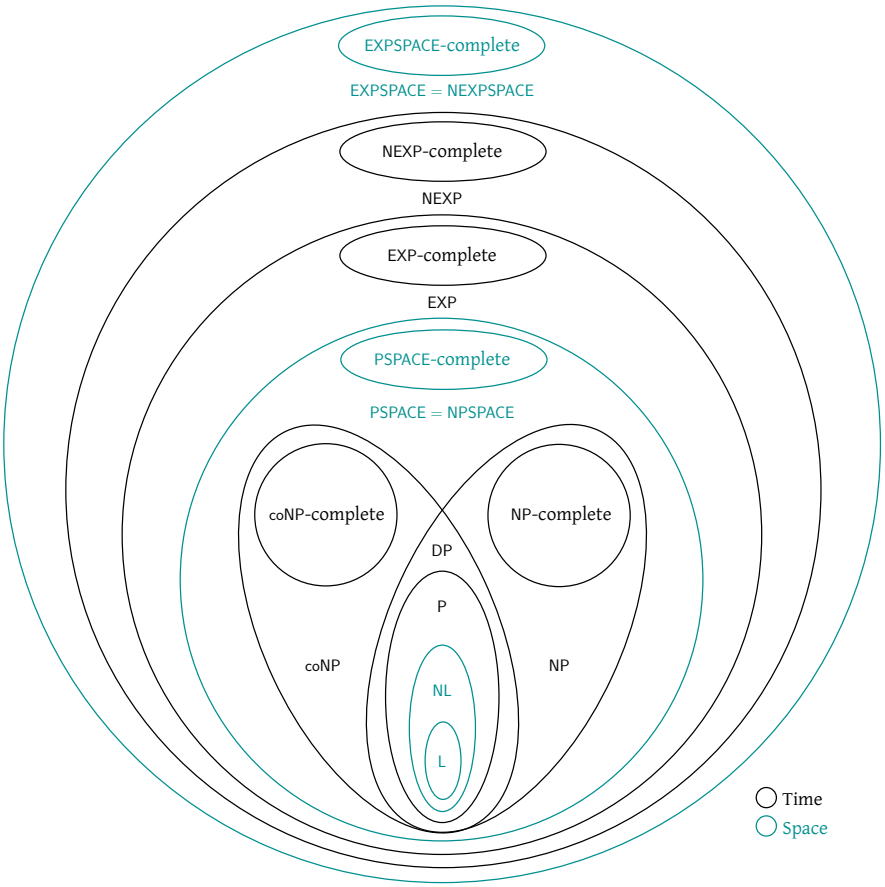
$$\text{INDEPENDENT-SET} = \{(G, k) : G \text{ is an undirected graph and } G \text{ contains an independent set of at least } k \text{ vertices}\} \text{ is NP-complete}$$

$$\text{INDEPENDENT-SET-SUCCINCT} = \{(C_G, k) : C_G \text{ is a SCR of an undirected graph } G \text{ and } G \text{ contains an independent set of at least } k \text{ vertices}\} \text{ is NEXP-complete}$$

The class NEXP-complete can be seen as a class of hard problems that are "easy" to describe.

We have not found a problem that is NP-complete for natural inputs but not NEXP-complete for succinct ones (similarly for other complexity classes).

6.7 Complexity Classes Summary Diagram



Bibliography

Agrawal, Manindra, Neeraj Kayal, and Nitin Saxena (2004). "PRIMES is in P". In: *Annals of mathematics*, pp. 781–793.

Cormen, Thomas H. et al. (2009). *Introduction to Algorithms*. Third Edition. The MIT Press. ISBN: 0262033844, 9780262033848.

Diophantus, Alexandrinus, Pierre de Fermat, and Claude Gaspar Bachet (1670). *Diophanti Alexandrini Arithmeticorum libri sex et de numeris multangulis liber unus*. Tolosæ: Collège de Plessis-Sorbonne.

Eppstein, David (Retrieved August 2020). *Computational Complexity of Games and Puzzles*. Available at `https://www.ics.uci.edu/~eppstein/cgt/hard.html`.

— (Retrieved January 2019). *The Geometry Junkyard*. Available at `https://www.ics.uci.edu/~eppstein/junkyard/open.html`.

Erdős, Paul (1950). "Az $1/x_1 + 1/x_2 + \cdots + 1/x_n = a/b$ egyenlet egész számú megoldásairól (On a Diophantine Equation)". In: *Mat. Lapok.* 1. In Hungarian, pp. 192–210.

Galperin, Hana and Avi Wigderson (1983). "Succinct representations of graphs". In: *Information and Control* 56.3, pp. 183–198.

Hayes, Brian (2006). "Gauss's Day of Reckoning". In: *American Scientist* 94.3, pp. 200–205.

Heath, Thomas L. and Euclid (1956). *The Thirteen Books of Euclid's Elements, Books 1 and 2*. New York, NY, USA: Dover Publications, Inc. ISBN: 0486600882.

Hilbert, David et al. (1902). "Mathematical problems". In: *Bulletin of the American Mathematical Society* 8.10, pp. 437–479.

Ladner, Richard E. (Jan. 1975). "On the Structure of Polynomial Time Reducibility". In: *J. ACM* 22.1, pp. 155–171. ISSN: 0004-5411. DOI: 10.1145/321864.321877. URL: http://doi.acm.org/10.1145/321864.321877.

Matiyasevich, Yuri Vladimirovich (1970). "The Diophantineness of enumerable sets". In: *Doklady Akademii Nauk*. Vol. 191. 2. Russian Academy of Sciences, pp. 279–282.

Pratt, Vaughan R (1975). "Every prime has a succinct certificate". In: *SIAM Journal on Computing* 4.3, pp. 214–220.

Rich, Elaine (2008). *Automata, computability and complexity: theory and applications*. Pearson Prentice Hall Upper Saddle River.

Robson, Eleanor (2001). "Neither Sherlock Holmes nor Babylon: A Reassessment of Plimpton 322". In: *Historia Mathematica* 28.3, pp. 167–206. ISSN: 0315-0860. DOI: https://doi.org/10.1006/hmat.2001.2317. URL: http://www.sciencedirect.com/science/article/pii/S0315086001923171.

Savitch, Walter J. (1970). "Relationships between nondeterministic and deterministic tape complexities". In: *Journal of computer and system sciences* 4.2, pp. 177–192.

Von Waltershausen, Wolfgang Sartorius (1856). *Gauss zum Gedächtniss*. S. Hirzel.

Wiles, Andrew (1995). "Modular elliptic curves and Fermat's last theorem". In: *Annals of mathematics* 141.3, pp. 443–551.

Yannakakis, C. and Papadirnitriouand M. (1986). "A Note on Succinct Representations of Graphs". In: *Information and Control* 71, pp. 181–185.

Index